# A Pattern-based deadlock-freedom analysis strategy for concurrent systems*

Pedro Antonino Augusto Sampaio
Centro de Informática, Universidade Federal de Pernambuco, Recife, PE, Brazil
{prga2@cin.ufpe.br, acas@cin.ufpe.br}

Jim Woodcock
Department of Computer Science, University of York, York, UK
jim.woodcock@york.ac.uk

January 31, 2019

**Abstract**

Local analysis has long been recognised as an effective tool to combat the state-space explosion problem. In this work, we propose a method that systematises the use of local analysis in the verification of deadlock freedom for concurrent and distributed systems. It combines a strategy for system decomposition with the verification of the decomposed subsystems via adherence to behavioural patterns. At the core of our work, we have a number of CSP refinement expressions that allows the user of our method to automatically verify all the behavioural restrictions that we impose. We also propose a prototype tool to support our method. Finally, we demonstrate the practical impact our method can have by analysing how it fares when applied to some examples.

***Keywords***— CSP; model checking; refinement; local analysis; behavioural patterns; system decomposition; deadlock freedom

## 1 Introduction

A deadlock is a long-standing, common pathology of concurrent systems [1, 2]. It occurs when the system reaches a state where all its components are stuck. The importance of deadlock analysis is attested by the fact that deadlock freedom is often considered to be the first step towards correctness for distributed

*The EU Framework 7 Integrated Project COMPASS (Grant Agreement 287829) financed most of the work presented here. This work is partially funded by INES, grants CNPq/465614/2014-0 and FACEPE/APQ/0388-1.03/14. No new primary data was created as part of the study reported here.

and concurrent systems. Moreover, safety properties can be reduced to deadlock checking [3]. As with many properties of concurrent systems, deadlock verification can be severely affected by the state space explosion problem [4].

One common way to cope with the state space explosion problem is to use local analysis [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Instead of checking the entire state space of the concurrent system, the analysis of small combinations of components is carried out to determine whether a system is deadlock free. In fact, for some complex systems, a method using local analysis might be the only practicable option. Local analysis methods are usually incomplete in the sense that they either guarantee deadlock freedom or are inconclusive. The latter means they can neither show that the system deadlocks nor prove deadlock freedom. Traditional local analysis techniques consist of either fully automatic *a posteriori* verification methods, or guidelines to the design of a system that, if followed, guarantee deadlock freedom by construction. The former techniques do not provide any guidance on how system designers can avoid deadlocks, whereas the latter ones do not provide automatic ways of checking that the guidelines were correctly followed.

We propose a method that provides both guidelines to construct deadlock-free systems and a procedure for automatically checking that the guidelines have been correctly followed. This method embodies a notion of decomposition that can be used to prove deadlock freedom for systems with an acyclic communication topology. Moreover, it relies on three behavioural patterns to deal with cyclic-topology systems. These behavioural patterns restrict both the behaviour of components and the structure of the system. Both the decomposition and behavioural patterns rely on local behavioural analysis. The decomposition relies on the analysis of pairs of components of the system, whereas the behavioural patterns constrain the behaviour of individual components. So, this method is not hindered by the state space explosion problem. Nevertheless, its efficiency comes at the price of incompleteness: deadlock freedom can only be proved for systems that fall into our decomposition/pattern-adherence method.

This proposed method is based on prior works that explored local analysis for the verification of deadlock freedom for concurrent and distributed systems. In fact, both the decomposition strategy and two out of the three patterns presented have been proposed decades ago [5, 7]. Nevertheless, we introduce a CSP formalisation based on refinement expressions that can be automatically checked by a refinement checker. In prior works, the decompositions and pattern adherence are characterised in terms of semantic properties that a system must have [5, 7]. This characterisation forces the user of such methods to understand not only the formalism but also the subtleties of its semantic models. On the other hand, our characterisation based on refinement expressions, together with design guidelines and tool support, gives a more practical support for the system designer. More importantly, prior works do not suggest an automatic way to test whether a system has a given semantic property, whereas our refinement expressions can be automatically checked by a refinement checker like FDR [21]. Finally, we conduct some experiments to measure the efficiency gains on the analysis of some practical examples.

This work is a significant extension of two previous works [11, 12]. The new contributions of this paper are as follows.

- We present formal proofs that our adherence to communication patterns guarantees deadlock freedom.
- We propose a method that systematises the application of system decomposition and pattern-adherence checking strategy to ensure deadlock freedom for a system. This systematisation should guide the user in applying our method in practice.
- We analyse the computational complexity of the proposed method, illustrate its application to three systems, and compare the method with three other approaches to deadlock analysis.
- We implemented a prototype tool that supports the proposed systematisation, saving a lot of manual effort that would otherwise be required in applying our method.

This paper is organised as follows. Section 2 introduces the CSP notation, some of its semantic models and a theory of networks of processes, based on CSP, that we use to represent and reason about concurrent systems. In Section 3, we present our decomposition strategy and how it can ensure deadlock freedom for acyclic-topology systems. Section 4 presents the formalisation of three behavioural patterns that prevent deadlocks. In Section 5, we present our method and a tool to support it. This section also presents the results of some experiments we conduct to assess the efficiency of our method when compared to traditional *a posteriori* verification techniques. Section 6 introduces a series of related works and how they relate to ours. Finally, in Section 7, we present our concluding remarks.

# 2 Background

We present a brief introduction to CSP, including the main operators and semantic models used in this work. Then we introduce a notion of live-network model, which is basically a sequence of components that obey some relevant properties. Two CSP models, used as running examples, are also presented.

## 2.1 CSP

Communicating Sequential Processes (CSP) [22, 23, 24] is a notation used to model concurrent systems where processes interact by exchanging messages. In this notation, sequential processes can be combined using high-level parallel operators to create complex concurrent processes. The CSP notation used here is the machine-readable version called $CSP_M$, which is the standard version for encoding CSP processes by the FDR tool [21]. In the following, we informally introduce some operators of this language using two CSP systems that serve as running examples throughout this paper. Our first example introduces a ring-buffer system.

**Running Example 1.** A ring buffer with $NCELLS$ storage cells is a system that stores data in a *first-in-first-out* fashion and where its storage cells are written to in a cyclic way. Cells are organised as if they were part of a ring, and once some piece of data is written to a cell, the next piece of data will be written to the next cell on this ring, provided the next cell is available. Our system can store up to $N = NCELLS + 1$ pieces of data because it has an extra cache storage space. $NCELLS$ (and $N$) is a global constant that serves as a parameter for our model; by changing $NCELLS$, we can create an arbitrary-sized system with $NCELLS > 0$ many cells. We use a central controller (described by process `Controller(cache,size,top,bot)` whose parameters are initially 0) to manage input and output requests to the buffer. This process has four parameters: `cache` holds the next element to be output, `size` keeps track of how many cells are full, `top` and `bot` keep track of which cell is the top (i.e., beginning) and the bottom (i.e., end) of the buffer, respectively. The parameters of a process represent its internal state.

```
Controller(cache,size,top,bot) =
    size < N & Input(cache,size,top,bot)
    []
    size > 0 & Output(cache,size,top,bot)
```

The process `c & P` behaves like `P` if the condition `c` is true and like `STOP` if `c` evaluates to false, where `STOP` is the atomic process that does nothing and deadlocks. The process `P [] Q` represents the external choice of `P` and `Q`, that is, the behaviours of `P` and `Q` are initially offered and then either `P` or `Q` is chosen. We point out that an external choice between `P` and `STOP` behaves just as `P`, i.e., `P [] STOP` $=$ `P`. So, process `Controller` offers the choice of behaving as `Input` if $size < N$ and as `Output` if $size > 0$.

If the buffer is not full (i.e., $size < N$), the controller can receive and store some data as described by process `Input`.

```
Input(cache,size,top,bot) =
    input?x ->
        (size == 0 & Controller(x,1,top,bot)
        []
        size > 0 & write.top!x ->
            Controller(cache,size+1,(top+1)%NCELLS,bot))
```

The prefixed process `a -> P` initially offers the event `a` and after this event is performed it behaves as `P`. $CSP_M$ also proposes the notion of a *channel* that transmits data. A channel `ch` is associated to the type of data, say values in the set *datatype*, they transmit. So, a channel gives rise to a number of events each of which denotes the transmission of a different piece of data, that is, event `ch.x` where $x \in datatype$ denotes the transmission of value $x$. A channel `ch` can output `ch!x` and input `ch?x` values. Outputting `ch!x` simply creates event ch.x based on the value $x$, whereas the input operation ch?x binds the values of ch's datatype to $x$ (intuitively, this means that ch can receive/input any value associated with this channel). So, ch!x -> P behaves as a simple prefix, whereas ch?x -> P behaves like an external choice: each possible value $v$ for $x$ gives rise to a new branch for which $x = v$. Note that channels and their operations are just syntactic sugar over events. For instance, process Input initially inputs some value $v$ on channel input, and then it proceeds execution with $x = v$. The expression (top+1)%NCELLS stands for the increment of top modulo NCELLS.

If the buffer is not empty, the controller can output and update its state as described by process Output.

```
Output(cache,size,top,bot) =
    output!cache ->
        (size > 1 & (read.bot?x ->
            Controller(x,size-1,top,(bot+1)%NCELLS))
        []
        size == 1 & Controller(cache,0,top,bot))
```

The process Cell(id,0) describes the individual cells that build up the buffer's storage space. It holds some value which can be read (using channel read.id) and updated (using channel write.id).

```
Cell(id,val) =
    read.id!val -> Cell(id,val)
    []
    write.id?x -> Cell(id,x)
```

Our final system, given by process RingBufferBehaviour, runs our controller process in parallel with $NCELLS$ storage cell processes using the indexed version of CSP's alphabetised-parallel operator.

```
RingBufferBehaviour = || i : {0..NCELL} @ A(i) [P(i)]
```

where

- P(0) = Controller(0,0,0,0)
- A(0) = {|read, write, input, output|},
- P(i) = Cell(i,0) for $i \in \{1 \ldots NCELL\}$
- A(i) = {|read.i, write.i|} for $i \in \{1 \ldots NCELL\}$
  - In $CSP_M$, the extension operator $\{|e_1, \ldots, e_n|\}$ gives the events that extend the elements $e_i$. For instance, in this example, we have {|read|} gives {read.i.v $\mid i \in \{0 \ldots N-1\} \wedge v \in \{0,1\}|\}$, assuming cells store binary values $v$.

The parallel process `P [X||Y] Q` allows $P$ and $Q$ to freely perform events not in the set of events $X \cap Y$, but to perform an event in $X \cap Y$, $P$ and $Q$ must synchronise on it. Additionally, $P$ ($Q$) is only allowed to perform events in $X$ ($Y$). $X$ is called the alphabet of $P$. This parallel operator also has an indexed version `|| e : S @ [A(e)] P(e)`, where `A(e)` gives an alphabet and `P(e)` gives a CSP process. For this indexed version, all processes are put in parallel using their corresponding alphabet. Similar to the binary version of this operator, shared events require synchronisation by all processes having the event on their alphabet and the non-shared events can be performed freely by a process. `RingBufferBehaviour` ensures that components synchronise on shared events, namely, read and write events only occur when the controller and the cells cooperate. We formally define the parallel composition for this system when we later introduce our network model. ■

The second running example that we use describes the well-known asymmetric solution to the dining philosophers problem.

**Running Example 2.** In the dining philosophers setting, $N$ philosophers are trying to eat on a shared round table; $N$ is a constant that also serves as a parameter for our example/model. To do so, each of them must acquire a pair of forks: one on its left-hand side and another on its right-hand side. Philosophers share their right-hand fork with their right neighbour and their left-hand one with the left neighbour. If all philosophers acquire their forks in the same order, they might run into the following deadlock. Say that all philosophers acquire first their left-hand fork and then their right-hand one, then they might reach a state where all of them have acquired their left-hand fork and are waiting for their right-hand one to be released. A well-known solution to avoid this deadlock is to have an asymmetric philosopher that acquires forks in the opposite order.

We describe the behaviour of philosophers that acquire and release first their left-hand fork and then their right-hand one by process `Phil(id)`. On the other hand, asymmetric philosophers are described by `APhil(id)`. Event `pickup.i.j` (`putdown.i.j`) is used by philosopher $i$ to acquire (release) fork $j$. Functions `next(i)` and `prev(i)` yield $(i+1)\%N$ and $(i-1)\%N$, respectively.

```
Phil(id) =
    sit.id -> pickup.id.id -> pickup.id!next(id) -> eat.id ->
        putdown.id.id -> putdown.id!next(id) -> getup.id ->
            Phil(id)

APhil(id) =
    sit.id -> pickup.id!next(id) -> pickup.id.id ->
        eat.id -> putdown.id!next(id) -> putdown.id.id ->
            getup.id -> APhil(id)
```

A fork can be acquired by a philosopher which later releases it as described by process `Fork(id)`.

```
Fork(id) =
    [] i : {id,prev(id)} @ pickup.i.id -> putdown.i.id -> Fork(id)
```

The process `[] x : S @ P(x)` is the indexed version of the external choice operator. For $\texttt{S} = \{v_1, \ldots, v_{|S|}\}$ where $|S|$ gives the size of set $S$, this process is $\texttt{P(}v_1\texttt{) [] ... [] P(}v_{|S|}\texttt{)}$.

The system implementing the asymmetric solution, given by `APhilsBehaviour`, runs in parallel $N$ forks, $N-1$ philosophers and an asymmetric philosopher. It relies on the indexed version of CSP's alphabetised-parallel operator to ensure processes synchronise on shared events.

```
APhilsBehaviour = || i : {0..2N-1} @ A(i) [P(i)]
```

where

- `P(i) = Phil(i)` for $i \in \{0, \ldots, N-2\}$
- `A(i) =` $AlphaPhil(i)$ for $i \in \{0, \ldots, N-2\}$
  - $AlphaPhil(i) = \{\texttt{sit.i}, \texttt{pickup.i.i}, \texttt{pickup.i.next(i)}, \texttt{eat.i}, \texttt{putdown.i.i}, \texttt{putdown.i.next(i)}, \texttt{getup.i}\}$
- `P(N-1) = APhil(N-1)`
- `A(N-1) =` $AlphaPhil(N-1)$
- `P(i) = Fork(i)` for $i \in \{N, \ldots, 2N-1\}$
- `A(i) =` $(\{\texttt{pickup.i.i}, \texttt{pickup.prev(i).i}, \texttt{putdown.i.i}, \texttt{putdown.prev(i).i}\}, \texttt{Fork(i)})$

■

## 2.2 Denotational semantics

In order to reason about processes, CSP embodies a collection of mathematical models. In this work, we use the *stable failures* model, and the less conventional *stable revivals* model.

In the stable-failures model, a process is represented by a pair $(F, T)$ containing its stable failures and its finite traces, respectively. The traces of a process are represented by a set of all the finite sequences of visible events that this process can perform; this set is given by $traces(P)$. The stable failures of a process are represented by a set of pairs $(s, X)$, where $s$ is a trace and $X$ is a set of events that the process can refuse to do after performing the trace $s$. At the state where the process can refuse events in $X$, the process must not be able to perform an internal action, otherwise this state would be unstable and would not be taken into account in this model. The function $failures(P)$ gives the set of stable failures of process $P$. Hence, the representation of process $P$ in this model is given by the pair $(failures(P), traces(P))$.

Before introducing how to systematically calculate the traces and failures of a process, we introduce a few more constructs of the $CSP_M$ notation. Similarly to `STOP`, `SKIP` is the atomic process that does nothing and terminates successfully.

Another useful atomic process, mainly from a theoretical perspective, is `div`, which is the diverging process. $\Sigma$ is the universal set of visible events; the invisible event $\tau$ and the termination signal $\checkmark$ are not members of this set. The internal (non-deterministic) choice process `P |~| Q` offers either `P` or `Q` non-deterministically. The process `P ; Q` behaves initially as process `P` and, once `P` successfully terminates, it behaves as process `Q`.

The renaming process `P [[R]]`, where `R` is a set of pairs `a <- b`, offers a deterministic choice of events in `S` whenever `P` offers `a`, where $\texttt{S} = \{\texttt{b} \mid \texttt{(a <- b)} \in \texttt{R}\}$. The hidden process `P \ S` offers the events not in $S$ whenever `P` offers them. On the other hand, `P \ S` can perform a $\tau$, the silent event, whenever `P` can perform an event in `S`. The interrupt process `P /\ Q` behaves like `P` and at any point it can be interrupted in which case it behaves as `Q`.

The $CSP_M$ notation does not provide an explicit operator for recursion, but it allows one to use the name of the process in its definition. For instance, `P = a -> P` performs `a`, and then recurses, behaving as `P`. Even though a formal construct is not available for recursion, we can define it as an equation where the right-hand side is a process context depending on the definition of the process itself, e.g. $X = F(X)$. For the process `P` given above, we can define it as `P = F(P)`, where $F(X) = \texttt{a -> } X$. For the purpose of giving the semantics of a recursive process, we use this style of definition.

The functions *traces*$(P)$ and *failures*$(P)$ are calculated inductively based on the constructs of the CSP language. The clauses for calculating the *traces* are presented in Table 1, whereas the clauses for calculating the *failures* are depicted in Table 2. The semantics of a recursive process can be calculated, using the presented clauses, thanks to the following equivalence. For a recursive process $P = F(P)$, $P \equiv \bigsqcap\{F^n(\texttt{div}) \mid n \in \mathbb{N}\}$, where $\bigsqcap S$ is the distributed application of the operator `|~|` to the processes in $S$. This equivalence also holds for the *stable revivals* model, presented later.

We illustrate the calculation of these behaviours using our ring-buffer system.

**Running Example 1.** We illustrate the traces and stable-failures for the processes `Controller(0,0,0,0)` and `Cell(0,0)`. For the following *failures* sets, $(tr, S)$ is a shorthand for all pairs $(tr, X)$ such that $X \subseteq S$; this makes our examples more compact.

- $\mathit{traces}(\texttt{Controller(0,0,0,0)}) =$
  $\{\langle\rangle, \langle \mathit{input}.0\rangle, \langle \mathit{input}.1\rangle, \langle \mathit{input}.2\rangle, \langle \mathit{input}.0, \mathit{input}.0\rangle, \langle \mathit{input}.0, \mathit{input}.1\rangle,$
  $\langle \mathit{input}.0, \mathit{input}.2\rangle, \langle \mathit{input}.0, \mathit{output}.0\rangle, \langle \mathit{input}.1, \mathit{output}.1\rangle, \ldots\}$

- $\mathit{failures}(\texttt{Controller(0,0,0,0)}) =$
  $\{(\langle\rangle, \Sigma - \{\mathit{input}.0, \mathit{input}.1, \mathit{input}.2\}),$
  $(\langle \mathit{input}.0\rangle, \Sigma - \{\mathit{input}.0, \mathit{input}.1, \mathit{input}.2, \mathit{output}.0\}),$
  $(\langle \mathit{input}.1\rangle, \Sigma - \{\mathit{input}.0, \mathit{input}.1, \mathit{input}.2, \mathit{output}.1\}),$
  $(\langle \mathit{input}.2\rangle, \Sigma - \{\mathit{input}.0, \mathit{input}.1, \mathit{input}.2, \mathit{output}.2\})), \ldots\}$

- $\mathit{traces}(\texttt{Cell(0,0)}) =$
  $\{\langle\rangle, \langle \mathit{read}.0\rangle, \langle \mathit{write}.0\rangle, \langle \mathit{write}.1\rangle, \langle \mathit{write}.2\rangle, \langle \mathit{read}.0, \mathit{write}.0\rangle,$
  $\langle \mathit{read}.0, \mathit{write}.1\rangle, \langle \mathit{read}.0, \mathit{write}.2\rangle, \langle \mathit{read}.0, \mathit{write}.0, \mathit{read}.0\rangle, \ldots\}$

- $\mathit{failures}(\texttt{Cell(0,0)}) =$
  $\{(\langle\rangle, \Sigma - \{read.0, write.0, write.1, write.2\}),$
  $(\langle read.0\rangle, \Sigma - \{read.0, write.0, write.1, write.2\}),$
  $(\langle write.0\rangle, \Sigma - \{read.0, write.0, write.1, write.2\}),$
  $(\langle write.1\rangle, \Sigma - \{read.1, write.0, write.1, write.2\})),$
  $(\langle write.2\rangle, \Sigma - \{read.2, write.0, write.1, write.2\}), \ldots\}$

■

In this model, the failures for a given trace are subset closed: if $(s, X) \in \mathit{failures}(P)$ then so is $(s, Y)$ provided $Y \subseteq X$. So, for some properties, we will be interested only in the maximal failures, considering the subset order, for each trace $s$. $\overline{\mathit{failures}}(P)$ denotes the set of such maximal failures for process $P$.

### Stable Revivals Model

In the stable revivals model, a process is described by a triple $(T, D, R)$ containing its traces, its deadlocks and its stable revivals, respectively. The deadlocks

| | |
|---|---|
| $\mathit{traces}(\texttt{STOP})$ | $= \{\langle\rangle\}$ |
| $\mathit{traces}(\texttt{SKIP})$ | $= \{\langle\rangle, \langle\checkmark\rangle\}$ |
| $\mathit{traces}(\texttt{div})$ | $= \{\langle\rangle\}$ |
| $\mathit{traces}(a \texttt{ -> } P)$ | $= \{\langle\rangle\} \cup \{\langle a\rangle \hat{}\, s \mid s \in \mathit{traces}(P)\}$ |
| $\mathit{traces}(P \texttt{ ; } Q)$ | $= (\mathit{traces}(P) \cap \Sigma^*) \cup \{s \hat{}\, t \mid s \hat{}\, \langle\checkmark\rangle \in \mathit{traces}(P) \wedge t \in \mathit{traces}(Q)\}$ |
| $\mathit{traces}(P \texttt{ [] } Q)$ | $= \mathit{traces}(P) \cup \mathit{traces}(Q)$ |
| $\mathit{traces}(P \texttt{ \|~\| } Q)$ | $= \mathit{traces}(P) \cup \mathit{traces}(Q)$ |
| $\mathit{traces}(P \texttt{ [\|} X \texttt{\|] } Q)$ | $= \bigcup\{s \underset{X}{\|} t \mid s \in \mathit{traces}(P) \wedge t \in \mathit{traces}(Q)\}$ |
| $\mathit{traces}(P \setminus X)$ | $= \{s \setminus X \mid s \in \mathit{traces}(P)\}$ |
| $\mathit{traces}(P \texttt{ [[R]]})$ | $= \{t \mid \exists\, s \in \mathit{traces}(P) \bullet s\, R^*\, t\}$ |
| $\mathit{traces}(P \,/\backslash\, Q)$ | $= \mathit{traces}(P) \cup \{s \hat{}\, t \mid s \in \mathit{traces}(P) \cap \Sigma^* \wedge t \in \mathit{traces}(Q)\}$ |

where

- $s \hat{}\, t$ represents the concatenation of sequences $s$ and $t$.
- $s \underset{X}{\|} t$ gives all the traces $w$ that are interleaving of $s$ and $t$ such that $w \upharpoonright X = s \upharpoonright X = t \upharpoonright X$.
- $s \upharpoonright X$ gives the trace resulting from removing events not in $X$ from $s$.
- $s \setminus X$ gives the trace resulting from removing events in $X$ from $s$.
- $\langle s_0, \ldots, s_n\rangle R^* \langle t_0, \ldots, t_m\rangle$ holds iff $n = m$ and $\forall\, i \in \{0 \ldots n\} \bullet s_i R t_i$.

Table 1: Semantic clauses for the traces model

$$
\begin{aligned}
&\mathit{failures}(\mathtt{STOP}) &&= \{(\langle\rangle, X) \mid X \subseteq \Sigma \cup \{\checkmark\}\}\\
&\mathit{failures}(\mathtt{SKIP}) &&= \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma \cup \{\checkmark\}\}\\
&\mathit{failures}(\mathtt{div}) &&= \emptyset\\
&\mathit{failures}(a \ \mathtt{->}\ P) &&= \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle \,\hat{}\, s, X) \mid (s, X) \in \mathit{failures}(P)\}\\
&\mathit{failures}(P\ \mathtt{;}\ Q) &&= \{(s, X) \mid s \in \Sigma^* \wedge (s, X \cup \{\checkmark\}) \in \mathit{failures}(P)\} \cup\\
& &&\quad \{(s \,\hat{}\, t, X) \mid s \,\hat{}\, \langle\checkmark\rangle \in \mathit{traces}(P) \wedge (t, X) \in \mathit{failures}(Q)\}\\
&\mathit{failures}(P\ \mathtt{[]}\ Q) &&= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathit{failures}(P) \cap \mathit{failures}(Q)\} \cup\\
& &&\quad \{(t, X) \mid (t, X) \in \mathit{failures}(P) \cup \mathit{failures}(Q) \wedge t \neq \langle\rangle\} \cup\\
& &&\quad \{(\langle\rangle, X) \mid X \subseteq \Sigma \wedge \langle\checkmark\rangle \in \mathit{traces}(P) \cup \mathit{traces}(Q)\}\\
&\mathit{failures}(P\ \mathtt{|\tilde{}|}\ Q) &&= \mathit{failures}(P) \cup \mathit{failures}(Q)\\
&\mathit{failures}(P\ \mathtt{[|}X\mathtt{|]}\ Q) &&= \bigcup\{(s \underset{X}{\parallel} t, Y \cup Z) \mid Y \upharpoonright (X \cup \{\checkmark\}) = Z \upharpoonright (X \cup \{\checkmark\}) \wedge\\
& &&\qquad\qquad\qquad (s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q)\}\\
&\mathit{failures}(P\ \backslash\ X) &&= \{(t \setminus X, Y) \mid (t, Y \cup X) \in failures(P)\}\\
&\mathit{failures}(P\ \mathtt{[[}R\mathtt{]]}) &&= \{(t, X) \mid (\exists\, t' \mid (t', t) \in R^* \wedge (t', R^{-1}(X)) \in \mathit{failures}(P)\}\\
&\mathit{failures}(P \mathbin{/\backslash} Q) &&= \{(s, X) \mid (s, X) \in \mathit{failures}(P) \wedge s \in \Sigma^* \wedge (\langle\rangle, X) \in \mathit{failures}(Q)\}\\
& &&\quad \cup \{(s, X) \mid s \,\hat{}\, \langle\checkmark\rangle \in \mathit{traces}(P) \wedge \checkmark \notin X\}\\
& &&\quad \cup \{(s \,\hat{}\, \langle\checkmark\rangle, X) \mid s \,\hat{}\, \langle\checkmark\rangle \in \mathit{traces}(P)\}\\
& &&\quad \cup \{(s \,\hat{}\, t, X) \mid s \in \mathit{traces}(P) \cap \Sigma^* \wedge (t, X) \in \mathit{failures}(Q)\}
\end{aligned}
$$

Table 2: Semantic clauses for the failures model

of a process are given by the set of traces after which the process refuses all the events in its alphabet; this set is given by *deadlocks*$(P)$. The stable revivals set is composed of triples $(s, X, a)$ containing a trace, a refusal set, and a revival event, respectively. The refusal set $X$, similarly to the one described in the stable failures model, describes the set of events that can be refused by the process after the trace $s$. The revival event $a$ represents an event that the process can offer after performing $s$ and refusing $X$. At the state where the revival is recorded, the process must not be able to perform an internal action, otherwise this state is unstable, not being taken into account. The function *revivals*$(P)$ gives the set of stable revivals of process $P$. Thus, the representation of a process $P$ in this model is given by $(traces(P), deadlocks(P), revivals(P))$. The necessity of this model comes from the fact that some properties that we intend to capture cannot be naturally captured using the notion of refinement over the failures model. Conflict freedom is a concrete example of such properties. Generally, requiring that different refusal behaviours $R_1$ and $R_2$, where $R_1 \subset R_2$, from a process based on whether a particular event is offered cannot be naturally captured using failures refinement; the subset-closed structure of refusal sets gets in the way of specifying such a property.

The functions *traces*$(P)$, *deadlocks*$(P)$ and *revivals*$(P)$ are calculated inductively based on the constructs of the CSP language. The clauses for calculating the *traces* are presented in Table 1. In the same way, the clauses for calculating *deadlocks* and *revivals* are depicted in Table 3 and Table 4, respectively.

$$
\begin{array}{ll}
\mathit{deadlocks}(\texttt{STOP}) & = \{\langle\rangle\} \\
\mathit{deadlocks}(\texttt{SKIP}) & = \emptyset \\
\mathit{deadlocks}(\texttt{div}) & = \emptyset \\
\mathit{deadlocks}(a \texttt{ -> } P) & = \{\langle a\rangle ^\frown s \mid s \in \mathit{deadlocks}(P)\} \\
\mathit{deadlocks}(P \texttt{ ; } Q) & = \{s \mid s \in \mathit{deadlocks}(P)\} \cup \{s ^\frown t \mid s ^\frown \langle \checkmark \rangle \in \mathit{traces}(P) \wedge t \in \mathit{deadlocks}(Q)\} \\
\mathit{deadlocks}(P \texttt{ [] } Q) & = ((\mathit{deadlocks}(P) \cup \mathit{deadlocks}(Q)) \cap \\
 & \quad \{s \mid s \neq \langle\rangle\}) \cup (\mathit{deadlocks}(P) \cap \mathit{deadlocks}(Q)) \\
\mathit{deadlocks}(P \texttt{ |\~{}| } Q) & = \mathit{deadlocks}(P) \cup \mathit{deadlocks}(Q) \\
\mathit{deadlocks}(P \texttt{ [|}X\texttt{|] } Q) & = \{u \mid \exists (s, Y) : \mathit{failures}(P)\, ; (t, Z) : \mathit{failures}(Q) \bullet \\
 & \qquad Y \setminus (X \cup \{\checkmark\}) = Z \setminus (X \cup \{\checkmark\}) \wedge u \in (s \underset{X}{\parallel} t) \cap \Sigma^* \wedge \Sigma^{\checkmark} = Y \cup Z\} \\
\mathit{deadlocks}(P \texttt{ \textbackslash{} } X) & = \{s \setminus X \mid s \in \mathit{deadlocks}(P)\} \\
\mathit{deadlocks}(P \texttt{ [[}R\texttt{]]}) & = \{s' \mid \exists s \bullet sRs' \wedge s \in \mathit{deadlocks}(P)\} \\
\text{where:} & \\
\quad \mathit{failures}(P) & = \{(s, X) \mid X \subseteq \Sigma^{\checkmark} \wedge s \in \mathit{Dead}\} \cup \{(s, X), (s, X \cup \{\checkmark\}) \mid (s, X, a) \in \mathit{Rev}\} \\
 & \quad \cup \{(s, X) \mid s ^\frown \langle \checkmark \rangle \in \mathit{Tr} \wedge X \subseteq \Sigma\} \cup \{(s ^\frown \langle \checkmark \rangle, X) \mid s ^\frown \langle \checkmark \rangle \in \mathit{Tr} \wedge X \subseteq \Sigma^{\checkmark}\}
\end{array}
$$

Table 3: Deadlocks semantic clauses

We illustrate the calculation of deadlocks and revivals using our ring-buffer system.

**Running Example 1.** We illustrate the revivals for the processes `Cell(0,0)` and `Controller(0,0,0,0)`. Since these two processes do not deadlock, they have empty *deadlocks* sets. For the following *revivals* sets, to make our presentation more compact, we use $(tr, S, S')$ as a shorthand denoting all pairs $(tr, X, a)$ such that $X \subseteq S$ and $a \in S'$.

- $revivals($`Controller(0,0,0,0)`$) =$
  $\{(\langle\rangle, \Sigma - \{input.0, input.1, input.2\}, \{input.0, input.1, input.2\}),$
  $(\langle input.0\rangle, \Sigma - \{input.0, input.1, input.2, output.0\},$
  $\{input.0, input.1, input.2, output.0\}),$
  $(\langle input.1\rangle, \Sigma - \{input.0, input.1, input.2, output.1\},$
  $\{input.0, input.1, input.2, output.1\}),$
  $(\langle input.2\rangle, \Sigma - \{input.0, input.1, input.2, output.2\},$
  $\{input.0, input.1, input.2, output.2\}), \ldots\}$

- $revivals($`Cell(0,0)`$) =$
  $\{(\langle\rangle, \Sigma - \{read.0, write.0, write.1, write.2\},$
  $\{read.0, write.0, write.1, write.2\}),$
  $(\langle read.0\rangle, \Sigma - \{read.0, write.0, write.1, write.2\},$
  $\{read.0, write.0, write.1, write.2\}),$
  $(\langle write.0\rangle, \Sigma - \{read.0, write.0, write.1, write.2\},$
  $\{read.0, write.0, write.1, write.2\}),$
  $(\langle write.1\rangle, \Sigma - \{read.1, write.0, write.1, write.2\},$
  $\{read.1, write.0, write.1, write.2\}),$
  $(\langle write.2\rangle, \Sigma - \{read.2, write.0, write.1, write.2\},$
  $\{read.2, write.0, write.1, write.2\}), \ldots\}$

$$
\begin{aligned}
&\mathit{revivals}(\mathtt{STOP}) &&= \emptyset\\
&\mathit{revivals}(\mathtt{SKIP}) &&= \emptyset\\
&\mathit{revivals}(\mathtt{div}) &&= \emptyset\\
&\mathit{revivals}(a\ \mathtt{->}\ P) &&= \{(\langle\rangle, X, a) \mid a \notin X\} \cup \{(\langle a\rangle \,\hat{}\, s, X, b) \mid (s, X, b) \in \mathit{revivals}(P)\}\\
&\mathit{revivals}(P\ \mathtt{;}\ Q) &&= \{(s, X, a) \mid\ \wedge (s, X, a) \in \mathit{revivals}(P)\}\\
& &&\quad \cup \{(s \,\hat{}\, t, X, a) \mid s \,\hat{}\, \langle \checkmark \rangle \in \mathit{traces}(P) \wedge (t, X, a) \in \mathit{revivals}(Q)\}\\
&\mathit{revivals}(P\ \mathtt{[]}\ Q) &&= \{(\langle\rangle, X, a) \mid (\langle\rangle, X) \in \mathit{failures}^b(P) \cap \mathit{failures}^b(Q)\\
& &&\qquad \wedge (\langle\rangle, X, a) \in \mathit{revivals}(P) \cup \mathit{revivals}(Q)\}\\
& &&\quad \cup \{(s, X, a) \mid (s, X, a) \in \mathit{revivals}(P) \cup \mathit{revivals}(Q) \wedge s \neq \langle\rangle\}\\
&\mathit{revivals}(P\ \mathtt{|{\sim}|}\ Q) &&= \mathit{revivals}(P) \cup \mathit{revivals}(Q)\\
&\mathit{revivals}(P\ \mathtt{[|}X\mathtt{|]}\ Q) &&= \{(u, Y \cup Z, a) \mid \exists\, s, t \bullet (s, Y) \in \mathit{failures}^b(P) \wedge (t, Z) \in \mathit{failures}^b(Q)\\
& &&\quad \wedge u \in s \mathbin{\underset{X}{\|}} t \wedge Y \setminus X = Z \setminus X\\
& &&\quad \wedge ((a \in X \wedge (s, Y, a) \in \mathit{revivals}(P) \wedge (t, Z, a) \in \mathit{revivals}(Q))\\
& &&\qquad \vee (a \notin X \wedge ((s, Y, a) \in \mathit{revivals}(P) \vee (t, Z, a) \in \mathit{revivals}(Q))))\}\\
& &&\quad \cup \{(u, Y \cup Z, a) \mid \exists\, s, t \bullet (s, Y, a) \in \mathit{revivals}(P) \wedge t \,\hat{}\, \langle \checkmark \rangle \in \mathit{traces}(Q)\\
& &&\quad \wedge Z \subseteq X \wedge a \notin X \wedge u \in s \mathbin{\underset{X}{\|}} t\}\\
& &&\quad \cup \{(u, Y \cup Z, a) \mid \exists\, s, t \bullet (t, Z, a) \in \mathit{revivals}(Q) \wedge s \,\hat{}\, \langle \checkmark \rangle \in \mathit{traces}(P)\\
& &&\quad \wedge Y \subseteq X \wedge a \notin X \wedge u \in s \mathbin{\underset{X}{\|}} t\}\\
&\mathit{revivals}(P\ \setminus\ X) &&= \{(s \setminus X, Y, a) \mid (s, Y \cup X, a) \in \mathit{revivals}(P)\}\\
&\mathit{revivals}(P\ \mathtt{[[}R\mathtt{]]}) &&= \{(s', X, a') \mid \exists\, s, a \bullet sRs' \wedge aRa' \wedge (s, R^{-1}(X), a) \in \mathit{revivals}(P)\}\\
&\text{where:} &&\\
&\quad \mathit{failures}^b(P) &&= \{(s, X) \mid X \subseteq \Sigma \wedge s \in \mathit{Dead}\} \cup \{(s, X) \mid (s, X, a) \in \mathit{Rev}\}
\end{aligned}
$$

Table 4: Revivals semantic clauses

■

The CSP framework offers, for each semantic model, a refinement relation between processes. `[F=` is the refinement relation for the stable failures model. `P [F= Q` holds if and only if $traces(Q) \subseteq traces(P)$ and $failures(Q) \subseteq failures(P)$ hold. This order relation can be seen as depicting that `Q` is more deterministic than `P`. `[V=` is the refinement relation for the stable revivals model. `P [V= Q` holds if and only if $traces(Q) \subseteq traces(P)$, $revivals(Q) \subseteq revivals(P)$ and $deadlocks(Q) \subseteq deadlocks(P)$ hold. This relation can be seen as depicting a finer more-deterministic order. While `P [F= Q` establishes that `Q` is more deterministic than `P` after each trace, `P [V= Q` establishes that `Q` is more deterministic than `P` for each event offered after each trace (namely, `Q` must refuse fewer events than `P` for each offer of an event $a$ after the trace $s$).

## 2.3 Network model

The concepts presented in this section are essentially a slight reformulation of concepts presented in [6, 5]. A *network* provides a model for a concurrent system in terms of its components.

**Definition 1.** A network is a sequence of components $\langle C_1, \ldots, C_n \rangle$, where $C_i = (A_i, P_i)$, $A_i \subseteq \Sigma$ and $P_i$ is a CSP process.

In this work, we consider only *live* networks. A network is live if and only if it is busy, non-terminating and triple disjoint. A network is busy if and only if every component is deadlock free, non-terminating if and only if every component does not terminate, and triple disjoint if and only if an event is shared by at most two components.

The *communication topology* (or topology for short) of a network can be analysed through its *communication graph.* It shows how components are connected, where a connection (i.e. edge) between two components represent that they (might) communicate/interact. This graph only depicts the (static) connections between components.

**Definition 2.** The communication graph of a network is an undirected graph where nodes denote components and there is an (undirected) edge between two nodes if and only if the corresponding components share some event.

For example, Figure 2 depicts the communication graph for the system implementing the (deadlocking) symmetric version of the dining philosophers problem with 3 philosophers and 3 forks, Figure 4 depicts the communication graph for an instance of our ring-buffer network with 3 storage cells, and Figure 6 depicts the communication graph for our (asymmetric) dining-philosophers network with 3 philosophers and 3 forks.

Note that this graph can be constructed based on a static analysis of components and their alphabets. The communication topology of a network plays an important role in deadlock analysis as we present later.

The behaviour of a network is given by a composition of the components' behaviours as follows.

**Definition 3.** Let $V = \langle C_1, \ldots, C_n \rangle$, where $C_i = (A_i, P_i)$, be a network. The behaviour of $V$ is given by the CSP expression: `||` $i$ `:` $\{1 \ldots n\}$ `@` `[`$A_i$`]` $P_i$

We (re-)define the systems in our running examples using the network model. Note how the behaviour of the following networks coincide with the processes that we have earlier introduced to capture the behaviour of the systems in our running examples. We define our ring buffer system as follows.

**Running Example 1.** Our ring-buffer system is defined by the `RingBuffer` network. Its behaviour requires processes to synchronise on shared events.

$$\texttt{RingBuffer} = \langle \mathit{Controller}, \mathit{Cell}(0), \ldots, \mathit{Cell}(N-1) \rangle$$

- $\mathit{Controller} = (\{|\texttt{read}, \texttt{write}, \texttt{input}, \texttt{output}|\}, \texttt{Controller(0,0,0,0)})$
- $\mathit{Cell}(i) = (\{|\texttt{read.i}, \texttt{write.i}|\}, \texttt{Cell(i,0)})$

- In $CSP_M$, the extension operator $\{|e_1, \ldots, e_n|\}$ gives the events that extend the elements $e_i$. For instance, in this example, we have $\{|\texttt{read}|\}$ gives $\{\texttt{read.i.v} \mid i \in \{0 \ldots N-1\} \wedge v \in \{0,1\}|\}$, assuming cells store binary values $v$.

■

The asymmetric solution to the dining philosophers problem is defined as follows.

**Running Example 2.** We define our asymmetric solution system using network `APhils`. Note that philosopher $N-1$ behaves asymmetrically. Also, we point out that this network's behaviour requires processes to synchronise on shared events.

$$\texttt{APhils} = \langle \mathit{Phil}(0), \ldots, \mathit{Phil}(N-2), \mathit{APhil}(N-1), \mathit{Fork}(0), \ldots, \mathit{Fork}(N-1)\rangle$$

- $\mathit{Phil}(i) = (\mathit{AlphaPhil}(i), \texttt{Phil(i)})$
  - $\mathit{AlphaPhil}(i) = \{\texttt{sit.i}, \texttt{pickup.i.i}, \texttt{pickup.i.next(i)}, \texttt{eat.i},$ $\texttt{putdown.i.i}, \texttt{putdown.i.next(i)}, \texttt{getup.i}\}$
- $\mathit{APhil}(i) = (\mathit{AlphaPhil}(i), \texttt{APhil(i)})$
- $\mathit{Fork}(i) = (\{\texttt{pickup.i.i}, \texttt{pickup.prev(i).i},$ $\texttt{putdown.i.i}, \texttt{putdown.prev(i).i}\}, \texttt{Fork(i)})$

■

To reason about the behaviour of a network, we define the notion of a *state*. A state presents an instant picture of the behaviour of the network in terms of its components' behaviours.

**Definition 4.** Let $V = \langle C_1, \ldots, C_n\rangle$ be a network where $C_i = (A_i, P_i)$. A state of the network is a pair $(s, R)$, where $R = (R_1, \ldots, R_n)$, such that:

- $s \in \Sigma^*$
- For all $i \in \{1 \ldots n\}$, $(s \upharpoonright A_i, R_i) \in \overline{\mathit{failures}}(P_i)$.

A network deadlocks if and only if it can reach a state in which no further action can be taken.

**Definition 5.** Let $V = \langle C_1, \ldots, C_n\rangle$ be a network where $C_i = (A_i, P_i)$, and $\sigma = (s, R)$, where $R = (R_1, \ldots, R_n)$, one of $V$'s states.

$$\mathit{Deadlocked}(\sigma) \mathrel{\widehat{=}} \mathit{Refusals}(\sigma) = \Sigma$$

where $\mathit{Refusals}(\sigma) \mathrel{\widehat{=}} \bigcup\{A_i \cap R_i \mid i \in \{1 \ldots n\}\}$

For live networks, ungranted requests are considered to be the building blocks of deadlocks. An ungranted request denotes a wait-for dependency from a component to another component in a given state. It arises, in a system state, when one component is offering an event which is being refused by its communication partner, so they cannot synchronise on this event. As components in a live network do not deadlock, a deadlock must be formed of a situation in which there exists a mutual wait between components.

**Definition 6.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\sigma = (s, R)$, where $R = (R_1, \ldots, R_n)$, one of $V$'s states. There is an ungranted request between $i$ and $j$ in state $\sigma$ if the following predicate holds.

$$\begin{aligned}&\mathit{ungranted\_request}(\sigma, i, j) \mathrel{\widehat{=}}\\ &\qquad \mathit{request}(\sigma, i, j) \wedge \mathit{ungrantedness}(\sigma, i, j) \wedge \mathit{in\_vocabulary}(\sigma, i, j)\end{aligned}$$

where:

- $\mathit{request}(\sigma, i, j) \mathrel{\widehat{=}} (A_i - R_i) \cap A_j \neq \emptyset$
- $\mathit{ungrantedness}(\sigma, i, j) \mathrel{\widehat{=}} (A_i - R_i) \cap (A_j - R_j) = \emptyset$
- $\mathit{in\_vocabulary}(\sigma, i, j) \mathrel{\widehat{=}} (A_i - R_i) \cup (A_j - R_j) \subseteq \mathit{Voc}$
  - $\mathit{Voc} = \bigcup_{i,j \in \{1 \ldots n\} \wedge i \neq j} (A_i \cap A_j)$ gives the vocabulary of the network, namely, the events requiring components to synchronise.

Given a fixed state $\sigma$, we use $i \rightarrow\!\bullet\, j$ to denote that there exists an ungranted request from $i$ to $j$ in $\sigma$. For a given fixed state, one can calculate all ungranted requests between components and create what we call a *snapshot graph*.

**Definition 7.** A snapshot graph for system state $\sigma$ is a directed graph where components are nodes and there is an edge from component $i$ to component $j$ if and only if there is an ungranted request in $\sigma$ from $i$ to $j$.

Unlike communication graphs that depict a static view of the (fixed) topology of the network, these snapshot graphs give instantaneous pictures of the dynamic behaviour of the system. Instead of attempting to capture the overall complexity of components' interactions, a snapshot graph depicts dependencies (i.e., ungranted requests) between components, which are the building blocks for our study of deadlock.

As mentioned, a network deadlocks when all components are blocked in a given state. For a live network, in such a state, all components must be waiting for some other component to advance. This situation implies that there must exist a cycle of ungranted requests between components. To be more concrete, the snapshot graph constructed for a deadlocked system state must exhibit a cycle (of dependencies). The following theorem, asserting these two facts, is our main tool in proving the soundness of our framework. These facts and their proofs can be found in [5, 7, 23]

**Theorem 1.** *Let $V = \langle C_1, \ldots, C_n \rangle$ be a network. In a deadlock state $\sigma$:*

1. *Each $C_i$ must be blocked.*
    - *A process $C_i$ is blocked in $\sigma$ if $A_i \subseteq Refusals(\sigma)$.*
2. *There must be a cycle of ungranted requests among components.*
    - *Given a state $\sigma$, a cycle of ungranted requests is a sequence $c \in \{1 \dots n\}^*$ such that for all $i$ in $\{1 \dots |c|\}$, $c_i \rightarrow\!\!\!\bullet\ c_{i \oplus 1}$ holds, where $\oplus$ denotes addition modulo $|c|$ and $|c|$ is the length of sequence/cycle $c$.*

In this paper, we will mainly prove that a system is deadlock free by showing that a cycle of ungranted requests cannot arise in any conceivable state of the system. Since such a cycle is a necessary condition for a deadlock, deadlock freedom can be proved this way. We finish this section by illustrating a few of the concepts we have presented.

**Example 3.** We discuss the concepts of communication and snapshot graph using the example of the symmetric (deadlocking) dining philosophers. This system is very similar to the asymmetric version that we have defined in Running Example 2 but instead of having one right-handed philosopher (as captured in component $APhil$), all philosophers are left-handed (as in component $Phil$). We discuss an instance of this system with 3 (left-handed) philosophers and 3 forks. Since all philosophers are left-handed, they first acquire their left-hand-side fork in order to eat. If all of them acquire their left-hand-side fork at the same time, let us call this system state $\sigma$, then all forks have been acquired and none of them can acquire their right-hand-side one; a deadlock occurs.

Figure 2 depicts the communication graph of this system (left-hand side) and the snapshot graph for system state $\sigma$ (right-hand side). On the snapshot graph, a dependency from $Fork_x$ to $Phil_y$ arises because the philosopher $y$ has acquired fork $x$ but has not released it yet. So, the fork is offering event *putdown.x.y* which is being refused by the philosopher, who is trying to acquire its right-hand-side fork. A dependency from $Phil_x$ to $Fork_y$ occurs because the philosopher $x$ is trying to acquire the fork $y$, which has already been acquired by the philosopher who is next in the cycle of dependencies ($Phil_{x \oplus 1}$, where $\oplus$ is addition modulo 3). The cycle of ungranted requests in the snapshot graph is an evidence of the deadlock system state $\sigma$ represents. Note that ungranted requests can only arise (in a snapshot graph) between components that are connected by an edge in the communication graph; if two components do not share an event, there cannot be an ungranted request between them. ■

# 3 Conflict freedom, acyclic networks and decomposition

Conflict freedom can be a very helpful property in proving deadlock freedom for a system. It can be used to decompose a proof of deadlock freedom for a system or, even, to prove that an acyclic network is deadlock free. In this

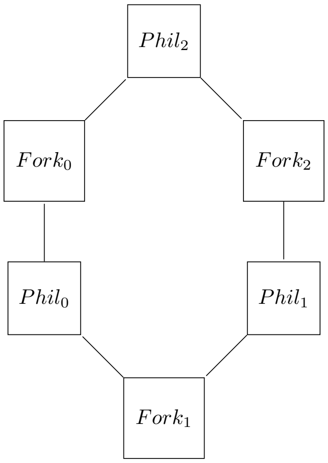


Figure 1: Communication graph for symmetric dining philosophers.

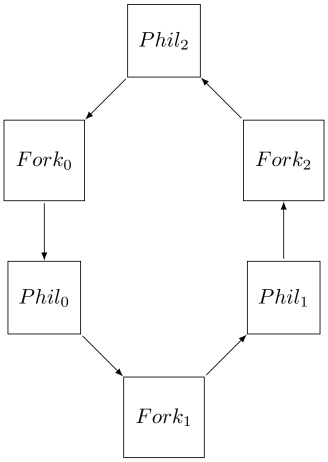


Figure 2: Communication graph and snapshot graph for symmetric dining philosophers.

section, we present a refinement expression that captures conflict freedom for a pair of components. An important advantage of this formalisation is that it can be automatically checked by a refinement checker. We also discuss how this property, and our refinement expression, can be used to break down a deadlock-freedom proof and to show an acyclic network deadlock free.

**Definition 8.** A *conflict* is a cycle of ungranted requests of size two, i.e. a cycle between a pair of components in a network. In a system state $\sigma$, a conflict between components $i$ and $j$ arises if and only if $i \rightarrow\!\!\!\bullet\, j$ and $j \rightarrow\!\!\!\bullet\, i$. In such a state, both components are willing to interact with one another, but they cannot agree on the event they need to synchronise on. Then, a pair of components $i$ and $j$ is *conflict free* if and only if in there is no system state in which a conflict between $i$ and $j$ occurs.

Conflict freedom can be more naturally captured by a refinement expression if the pair of components being verified is placed in a particular behavioural context. This context abstracts the behaviour of both components by using the process `Abs`. It abstracts away the events that components can perform individually as they do not play a part in making a system deadlock. This abstraction plays a fundamental role in our work; instead of their original behaviour, our behavioural analyses examine the abstract behaviour of components.

**Definition 9.** For a given network $V = \langle C_1, \ldots, C_n \rangle$, where $C_i = (A_i, P_i)$, we have that $\texttt{Abs(i)} = P_i \setminus (\Sigma - \mathit{Voc})$.

Our context is also designed so it offers the special event *req* whenever these abstract components can both offer an event from $A_i \cap A_j$. This context is given by the process `Context`.

**Definition 10.** Let $C_i$ and $C_j$ be two components of network $V$.

```
Context(i,j) = Ext(i,j)[union(A(i),req)||union(A(j),req)] Ext(j,i)
```

where `Ext(i,j) = Abs(i) [[ x <- x, x <- req | x <- inter(A(i),A(j))]]`

When in this context, if both components are making requests to each other (i.e. they are both offering events in $A_i \cap A_j$) but they do not agree on this event (i.e. they offer different events), then they both can offer *req* so they can synchronise on *req* but they cannot synchronise on any event on $A_i \cap A_j$. So, a conflict arises when the *req* event is offered and $A_i \cap A_j$ is refused. Hence, a conflict free pair of processes does not have some revival of the form $(s, A_i \cap A_j, req)$. So, the characteristic process `ConflictFreeSpec` capturing conflict freedom must have all possible revivals but these ones.

**Definition 11.** Let $C_i$ and $C_j$ be two components in network $V$.

```
ConflictFreeSpec(i,j) =
    let U_A = union(A(i),A(j))
        I_A = inter(A(i),A(j))
        CF = ((|~| ev : I_A @ ev -> CF)
                 [] req -> CHAOS(union(U_A,{req})))
             |~|
             (|~| ev : U_A @ ev -> CF)
within CF
```

where: `CHAOS(A) = SKIP |~| STOP |~| (|~| ev : A @ ev -> CHAOS(A))`

The following theorem depicts the refinement expression we propose to check conflict freedom. Note that we use the stable revivals model, as this property can be more intuitively captured in this model. The reason is the nature of conflict freedom. A pair of processes are conflict free if they are not at all willing to engage or if they are willing and able to engage. This implies that in the stable failures model, we would need a process that could refuse all shared events as well as offer some events to engage, but this intuitively violates the property that refusals should be subset closed.

**Theorem 2.** `ConflictFreeSpec(i,j) [V= Context(i,j)` $\Rightarrow$ *the pair of components* $(C_i, C_j)$ *is conflict free.*

*Proof.* In a conflict free state, the `Context` process must not have a revival of the form $(s, X, req)$ where $A_i \cap A_j \subseteq X$. After calculation of the revivals of the `ConflictFreeSpec`, its revivals are given by the following set comprehension expression $\{(s, X, a) | s \in (A_i \cup A_j \cup \{req\})^* \wedge a \in (A_i \cup A_j \cup \{req\}) \wedge a \notin X \wedge (a = req \Rightarrow (A_i \cap A_j) \not\subseteq X)\}$; this specification has all the possible revivals but the ones generated by a conflict. If the refinement expression holds, then $\mathit{revivals}(\texttt{ConflictFreeSpec(i,j)}) \supseteq \mathit{revivals}(\texttt{Context(i,j)})$. Hence, in this case `Context` has only conflict free revivals. For the other components of this model, *deadlocks* and *traces*, the restrictions are evident. Traces are not restricted at all, $\mathit{traces}(\texttt{ConflictFreeSpec(i,j)}) = (A_i \cup A_j \cup \{req\})^*$, also as deadlock can only arise if there is a conflict, we restrict the set of deadlocks to be empty, $\mathit{deadlocks}(\texttt{ConflictFreeSpec(i,j)}) = \emptyset$. □

Conflict freedom can be used to break down the verification of deadlock for a network to the analysis of some of its subnetworks. In the communication graph of a network, the *disconnecting edges* are the edges whose removal would increase the number of connected components in this graph – these are bridges in graph-theoretic terms. We call essential subnetworks the connected components resulting after removing some of these edges.

**Theorem 3** (Theorem 4 in [6]). *A network $V$ is deadlock free if the essential subnetworks resulting from the removal of conflict-free disconnecting edges are deadlock free. A disconnecting edge is conflict free if and only if the two components participating on it are conflict free.*

We give an example to illustrate the concepts linked to decomposition.

**Example 4.** Let $V = \langle C_0, C_1, C_2, C_3, C_4, C_5\rangle$ be a live network for which communication graph is given in Figure 3. This network has two rings ($C_0, C_1, C_2$ and $C_3, C_4, C_5$) which are interconnected via components $C_0$ and $C_3$. Also, let $\sigma_1$, $\sigma_2$, and $\sigma_3$ be states of this network for which snapshot graphs are also depicted in Figure 3.

This network has a single disconnecting edge $(C_0, C_3)$. Note that by removing this edge, we end up with two essential subnetworks (i.e. connected components in graph-theoretic terms) $\langle C_0, C_1, C_2\rangle$ and $\langle C_3, C_4, C_5\rangle$. If, instead, we decided to remove any other edge, we would end up with a single connected component. Hence, all other edges are not disconnecting.

In a live network, a component is either blocked because it is in a path of ungranted requests leading to a blocked subnetwork or because it is in a cycle of ungranted requests; such a cycle is sort of a fundamental blocked subnetwork. Considering our network, a deadlocked state could arise because there is a conflict between our two rings, i.e. a conflict between $C_0$ and $C_3$, as for instance in state $\sigma_1$. Note that in this state, all other components depend on this pair of components to progress. If we remove the $C_0, C_3$ edge (from the communication graph) and analyse the two rings independently, these two separate subnetworks could even be deadlock free and still admit exactly the paths of ungranted requests leading to the conflict shown in $\sigma_1$ when put together. Note that these paths on their own are not blocking either ring (hence, deadlock free could admit these paths), the conflict is the root cause of the deadlock. Therefore, inadvertently removing disconnecting edges and might lead to unknowingly removing the root cause of a deadlock from our analysis. Disconnecting edges can only be removed if they are conflict free.

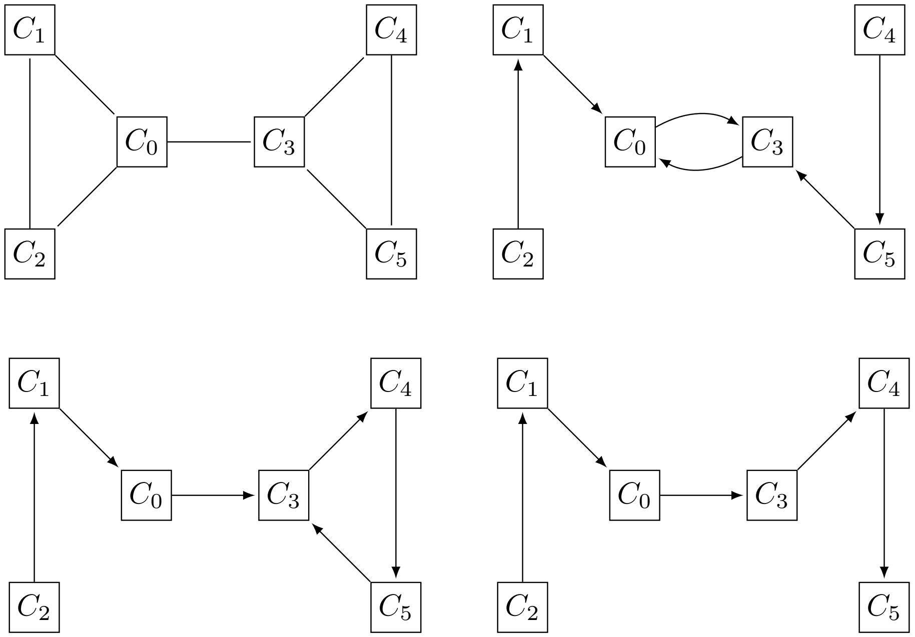


Figure 3: Communication graph and snapshot graphs for states $\sigma_1$, $\sigma_2$, and $\sigma_3$, respectively (left to right, top to bottom), for our examples.

Let us assume now that the edge $C_0, C_3$ is conflict free (so state $\sigma_1$ is unreachable). For a deadlock to arise, it must be that one of the rings is blocked and the components in the other ring are in ungranted-request paths leading to it. This happens, for instance, in state $\sigma_2$ where we have that the subnetwork $\langle C_3, C_4, C_5 \rangle$ is blocked by a cycle of dependencies and the other ring (involving $C_0, C_1, C_2$) depends on this subnetwork, so we have a deadlock. As our disconnecting edge is conflict free, we could analyse our rings independently. This state shows, however, that it only takes one blocked (essential) subnetwork to make a system deadlock. Note here that the path in $\sigma_2$ around ring $C_0, C_1, C_2$ is a valid configuration of a deadlock free (sub)network. The cycle of ungranted requests around the ring $C_3, C_4, C_5$, however, means that this subnetwork deadlocks.

If the edge $C_0, C_3$ is conflict free and the two rings are independently deadlock free, it is impossible for one ring to be blocked by the other. State $\sigma_3$ shows a state where ring $C_0, C_1, C_2$ depends on a *progressing* ring $C_3, C_4, C_5$. ■

Thus, our refinement expression can be used to show that a disconnecting edge is conflict free, enabling one to decompose the network into smaller essential subnetworks. Also, note that the identification of disconnecting edges can be carried out statically, i.e., by examining the communication graph, so generally this should be considerably simpler than showing conflict freedom for them. Note that a given network has a unique set of conflict-freedom disconnecting edges that can be removed to decompose the network.

In addition to that, from this theorem, we can deduce the following corollary:

**Corollary.** *A (live) conflict-free acyclic (topology) network must be deadlock free.*

A network is conflict free if and only if all its edges are conflict free. Note that for an acyclic network, all edges are disconnecting ones. So, provided that all edges are conflict free, we can remove them and, as a result, we would have essential subnetworks with a single component. Thus, as components are deadlock-free, by the busyness requirement, this acyclic network must be deadlock free.

So, using our refinement expression, one can systematically decompose a network or even prove deadlock freedom for conflict-free acyclic networks. Both these applications can substantially reduce the complexity of deadlock-freedom analysis at a fairly low price; our conflict analysis only involves the examination of pairs of components as opposed to the system's overall behaviour. For instance, a conflict-free acyclic network can be simply ensured deadlock free by this sort of pairwise (local) analysis; we illustrate this with an example.

**Running Example 1.** Our ring-buffer network can be checked deadlock free by using decomposition alone. In this example, we analyse a network with one controller and three storage cells. In Figure 4, we depict the communication graph of our example system and which sort of conflicts could potentially happen (they do not actually happen as we discuss next). $Contr$ represents the controller component, whereas $Cell_i$ depicts a $Cell(i)$ component. This system has an acyclic communication graph (i.e. topology) so every edge is disconnecting. Moreover, every edge (i.e. pair of components connected by an edge) is conflict free: whenever the controller wants to read from or write to a cell, it can do so. So, none of the possible conflicts depicted in Figure 4 can arise in any given system state. As all disconnecting edges are conflict free, we can decompose this network by removing all edges. This process results in 4 essential networks all of which have a single component. Since all components are deadlock free by virtue of our network being live, these essential subnetworks are deadlock free. Finally, by Theorem 3, this network must be deadlock free. ■

# 4 Behavioural patterns

Despite being useful, conflict-freedom testing has its limitation. For instance, it is unable to show deadlock freedom for cyclic-topology systems or even to decompose systems with no disconnecting edges. For these cases, we propose pattern adherence as an alternative effective verification technique that relies on local (compositional) analysis to ensure deadlock freedom. Once again, we give up completeness in exchange for efficiency. We can only ensure deadlock freedom for systems that adhere to one of the communication/behavioural patterns that we propose but adherence to these patterns can be efficiently tested in a local/compositional way.

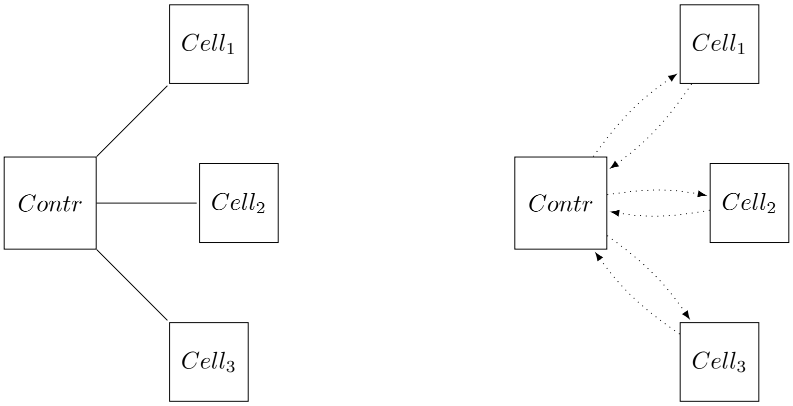


Figure 4: Communication graph for RingBuffer network with 3 storage cells and example of conflicts that could arise, respectively.

In this section, we present a characterisation of behavioural patterns using refinement expressions that can prove deadlock freedom for some networks with an arbitrary communication topology. We introduce our formalisation and a proof of their soundness. We formalise requirements on the behaviour of components as refinement assertions. This formalisation permits automatic checking of behavioural constraints using refinement checkers, providing their model sizes are tractable.

## 4.1 Resource allocation

The resource allocation pattern can be applied to systems that, in order to perform an action, have to acquire some shared resources. In this pattern, the components of a network are divided into users and resources. A user represents a component of the system that needs to acquire some resources in order to fulfil its final purpose. A resource is at the disposal of the users of the system.

As with design patterns for object-oriented programming languages, our behavioural patterns are specified in terms of some distinguished elements. For instance, when designing a resource allocation network, some components are meant to be users, whilst others are meant to be resources. These pattern elements are identified through a *pattern descriptor*.

A resource-allocation descriptor for a network $V$, with $n$ components and $\Sigma$ as alphabet, is a tuple $\mathcal{M} = (\mathcal{C}, acquire, release)$ containing a set $\mathcal{C} \subseteq \{1 \ldots n\} \times \{1 \ldots n\}$ and two functions *acquire* and *release*. Each pair $(i, j) \in \mathcal{C}$ represents the existence of a connection in $V$ between the user component $i$ and the resource component $j$. The function application $acquire(i, j)$ ($release(i, j)$) gives the event used by $i$ to acquire (release) $j$. These functions must be defined to all pairs in $\mathcal{C}$. As conventions, $users \mathrel{\widehat{=}} \{i \mid \exists\, j : \{1 \ldots n\} \bullet (i, j) \in \mathcal{C}\}$, $resources \mathrel{\widehat{=}} \{j \mid \exists\, i : \{1 \ldots n\} \bullet (i, j) \in \mathcal{C}\}$, $resources(i) \mathrel{\widehat{=}} \{j \mid (i, j) \in \mathcal{C}\}$, $users(j) \mathrel{\widehat{=}} \{i \mid (i, j) \in \mathcal{C}\}$.

A network and a resource allocation descriptor are compliant with the resource allocation pattern if they fulfil some structural and behavioural conditions. The structural constraint restricts the static elements of the network. For instance, it may restrict which connections can be made between components or which events are shared amongst components. On the other hand, behavioural constraints restrict the behaviour of the components of the network.

The structural constraint for this pattern requires the identification of elements as either resources or users. Additionally, it restricts which events are shared. In this case, only events for acquisition and release of resources can be shared. This constraint, which appears recurrently in our patterns, singles out which events are used for interaction between components. Therefore, we can restrict the behaviour of components on these events to avoid deadlocks.

**Definition 12.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\mathcal{M}$ a resource allocation pattern descriptor for $V$. The network $V$ and the descriptor $\mathcal{M}$ are structurally compliant if and only if the following predicates hold.

- $\mathit{partitioned} \mathrel{\widehat=} \{1 \ldots n\} = \mathit{users} \cup \mathit{resources} \wedge \mathit{users} \cap \mathit{resources} = \emptyset$
- $\mathit{mutually\_disjoint\_events} \mathrel{\widehat=}$
  $\neg \exists\, i : \mathit{users}\, ;\, j : \mathit{resources} \bullet \mathit{acquire}(i,j) = \mathit{release}(i,j)$
- $\mathit{controlled\_alpha\_users} \mathrel{\widehat=}$
  $\forall\, i : \mathit{users} \bullet A_i \cap \mathit{Voc} = \{\mathit{acquire}(i,j), \mathit{release}(i,j) \mid j \in \mathit{resources}(i)\}$
- $\mathit{controlled\_alpha\_resources} \mathrel{\widehat=}$
  $\forall\, i : \mathit{resources} \bullet A_i \cap \mathit{Voc} = \{\mathit{acquire}(j,i), \mathit{release}(j,i) \mid j \in \mathit{users}(i)\}$

On the behavioural side, we present two CSP processes that define the expected behaviour of a user component and of a resource component. The resource component should offer the events of acquisition to all users able to acquire this resource and, once acquired, it offers the release event to the user that has acquired it.

**Definition 13.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network, and $\mathcal{M}$ a resource allocation descriptor for $V$. $\mathit{ResourceSpec}(i)$ defines the expected behaviour of a resource component.

```
ResourceSpec(i) = [] j : users(i) @ acquire(j,i) ->
    release(j,i) -> Resource
```

A user component should first acquire all the necessary resources, and then release them. Both acquiring and releasing must be performed using the order denoted by the $\mathit{order}(i)$ sequence.

**Definition 14.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network, $\mathcal{M}$ a resource allocation descriptor for $V$, and $\mathit{order}(i)$ a function giving the sequence in which resources are acquired by component $i$. $\mathit{UserSpec}(i)$ defines the expected behaviour of a user component.

```
UserSpec(i) =
    let Acquire(s) =
            if s != <> then acquire(i,head(s)) -> Acquire(tail(s))
            else SKIP
        Release(s) =
            if s != <> then release(i,head(s)) -> Release(tail(s))
            else SKIP
        User(s) = Acquire(s);Release(s);User(s)
    within User(order(i))
```

To ensure that a component meets its specification, the behavioural constraint requires the stable failure refinement relation to hold between the specification and the behaviour of a component.

**Definition 15.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, $\mathcal{M}$ a resource allocation pattern descriptor for $V$, $\mathit{order}(i)$ a function giving the sequence in which resources are acquired by component $i$, and $>_{RA}$ a strict total order on resources. $V$ and $\mathcal{M}$ are behaviourally compliant if and only if the following hold.

- $\forall\, i : \mathit{users} \bullet$ `UserSpec(i) [F= Abs(i)`
- $\forall\, i : \mathit{resources} \bullet$ `ResourceSpec(i) [F= Abs(i)`
- $\forall\, i : \mathit{users} \bullet \mathit{order}(i)$ must respect $>_{RA}$

A sequence $\langle s_1, \ldots, s_n \rangle$ *respects* an order $>$ if the elements in the sequence are ordered respecting $>$, namely, for all $i \in \{1 \ldots n\}$ we have that $s_i > s_{i+1}$.

Note that we require an abstract version of a component's behaviour to comply with its specification. The reason is that, to guarantee deadlock freedom, we only need to regulate the behaviour related to events used in the interaction between components. The behaviour of a component on non-shared events is not relevant in deadlock analysis, as the component can perform them individually. So, in the analysis of deadlock freedom, we can study the network composed of the abstract behaviours of components, rather than the fully detailed network. This result is presented in the following lemma.

**Lemma 1.** *Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $V' = \langle C'_1, \ldots, C'_n \rangle$ another network where $C'_i = (A_i,$ `Abs(i)`$)$; `Abs(i)` as per Definition 9. If $V'$ is deadlock free then so is $V$.*

*Proof.* We prove this claim by contradiction. Assuming that $V'$ is deadlock free and $V$ is not, we reach a contradiction. Let us assume that $\sigma = (s, (R_1, \ldots, R_n))$ is a deadlock state of $V$, thus $Refusal(\sigma) = \Sigma$. In $\sigma$, none of the components of $V$ must be willing to perform an event that is not in the vocabulary, that is, $\overline{Voc} \subseteq R_i$ for all $i \in \{1 \ldots n\}$. If that was not the case, then $\sigma$ would not be a deadlocked state. Hence, from the definition of a network state and from the clause calculating the failures for the hiding operator, we can deduce that the state $\sigma' = (s \upharpoonright Voc, (R_1, \dots, R_n))$ is a valid state for $V'$. So, since $Refusal(\sigma) = Refusal(\sigma')$ and both networks have the same alphabet, then $\sigma'$ represents a deadlock for $V'$, thus a contradiction. $\Box$

As the main result of this section, we show that compliance to the resource-allocation pattern guarantees deadlock freedom. It ensures that resources in a path of ungranted requests respect our strict order $>_{RA}$, namely, if there is a path from $r_1$ to $r_n$ then $r_1 >_{RA} r_n$. Hence, a cycle of ungranted requests would lead to a contradiction in the form of $r >_{RA} r$. Therefore, such cycles cannot arise and that, in turn, guarantees deadlock freedom. This sort of coincidence between paths of ungranted requests and a strict order is a core common idea shared by our patterns which makes them sound. Note that the idea of ordering resources and their acquisition to prevent deadlocks, which inspired ours and many other works, reaches back decades [1, 25].

**Theorem 4.** *Let $V = \langle C_1, \dots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, $\mathcal{M}$ a resource allocation pattern descriptor for $V$, $order(i)$ a function giving the sequence in which resources are acquired by component $i$, and $>_{RA}$ a strict total order on resources. If $V$ and $\mathcal{M}$ are resource allocation compliant then $V$ is deadlock free.*

*Proof.* We prove this theorem by showing that the network $V' = \langle C'_1, \dots, C'_n \rangle$, where $C'_i = (A_i, \texttt{Abs(i)})$, is deadlock free and by using Lemma 1.

To prove that $V'$ is deadlock free, we rely on the second condition of Theorem 1. So, we show that there cannot be a cycle of ungranted requests between components of this network.

First, given that *partitioned* holds, we know that a component must be either a resource or a user. Moreover, thanks to *mutually_disjoint_events*, we know that events cannot be used for both acquiring and releasing a resource. Conditions *controlled_alpha_users*, *controlled_alpha_resources* and triple-disjointness implies that no two resources, nor two users, can share an event. As no two resources, nor two users, can share an event, the predicate *request* cannot be met and, as a consequence, there cannot be an ungranted request between such two elements. Thus, a cycle of ungranted requests in this network must be composed of alternating users and resources. So, we move on to analyse the interaction between a user and a resource.

Let $C_r$ be a resource component and $C_u$ a user one. From the required behaviour compliance, we know that the `Abs(i)` has to behave exactly as `User(u)` or `Resource(r)` for $i = u$ or $i = r$, respectively. So, we can analyse the behaviour of `Abs(i)` in terms of the behaviour of these two processes.

Based on the behaviour of `User(u)` and `Resource(r)`, we know that an ungranted request can only arise from $u$ to $r$ in a state $\sigma$ if and only if $u$ is ready to acquire $r$, but $r$ has already been acquired by another user. In all other cases, $u$ and $r$ can successfully interact preventing the ungranted request. Note, then, that a cycle of ungranted requests can only involve resources that have already been acquired. Thus, we only discuss paths and cycles of ungranted request where all resources have been acquired. Additionally, based on `User(u)`'s

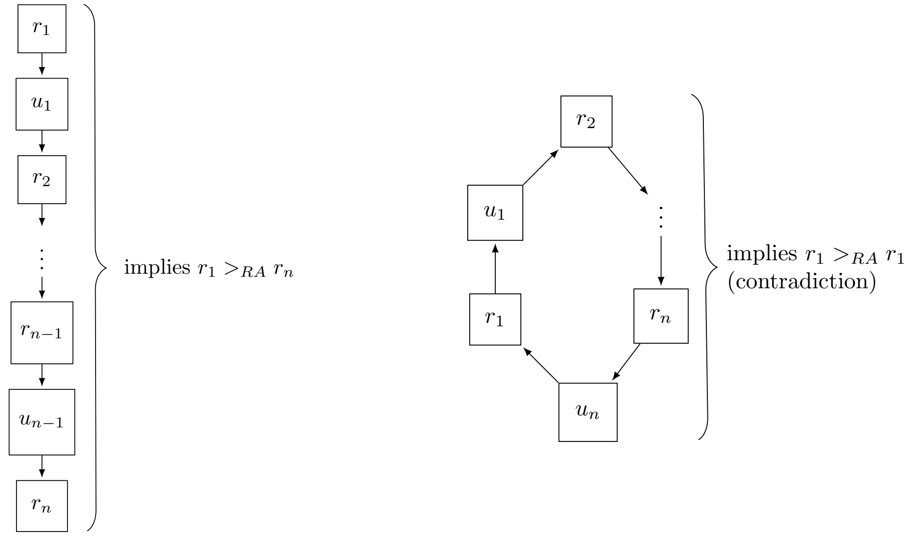


Figure 5: Chain of ungranted requests and cycle of ungranted requests coincidence with $>_{RA}$.

behaviour, we know that (i) $u$ is trying to acquire a resource that is higher, considering $>_{RA}$, than any of its acquired resources.

Two kind of ungranted requests can happen from a resource $r$ to one of its users $u$. An ungranted request from $r$ to $u$ might arise if either $r$ has not been acquired yet or $r$ has been acquired by $u$ but $u$ is not yet ready to release it. We are only interested in the later since the former case cannot be part of a cycle of ungranted requests; note that a free resource cannot be the target of an ungranted request, as the user issuing the request to acquire this resource would just be able to do so (i.e. the request would be "granted").

So, we have that cycles of ungranted requests can only be formed by chains of the form $r \rightarrow\!\!\bullet\ u \rightarrow\!\!\bullet\ r'$ where $r$ has been acquired by $u$ and $u$ is trying to acquire $r'$. Such a chain implies that $r >_{RA} r'$ by (i). So, for any pair of resources $r_1$ and $r_n$ in a path of ungranted requests $r_1 \rightarrow\!\!\bullet\ u_1 \rightarrow\!\!\bullet\ r_2 \rightarrow\!\!\bullet \ldots \rightarrow\!\!\bullet\ r_{n-1} \rightarrow\!\!\bullet\ u_{n-1} \rightarrow\!\!\bullet\ r_n$, it must hold that $r_1 >_{RA} r_n$.

Note, then, that the existence of a cycle of ungranted requests would lead to a contradiction. Such a cycle is a resource-user chain of ungranted requests that begins and ends in the same resource. That would imply that a reflexive pair $(r, r)$ belongs to $>_{RA}$, contradicting the fact that $>_{RA}$ is a strict order on resources. In Figure 5, we illustrate the coinciding of the order in which resources appear in paths of ungranted requests with the order $>_{RA}$, and the contradiction it leads to in the context of cycles of ungranted requests.

□

We use our asymmetric dining philosophers example to illustrate how patterns can be applied to ensure deadlock freedom.

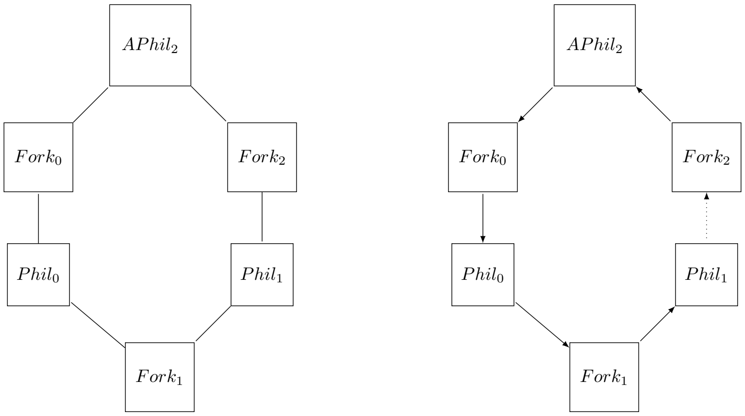


Figure 6: Communication graph and snapshot graph for asymmetric dining philosophers.

**Running Example 2.** Our asymmetric-dining-philosophers network does not have an acyclic topology. It is in fact a large ring of alternating fork and philosopher components. Hence, decomposition is not an option as all edges are not disconnecting. Note that by removing any edge the number of connected (graph-theoretic) components/essential subnetworks does not increase; we always have a single subnetwork. In this example, we analyse a network with 3 forks and 3 philosophers. Figure 6 depicts the communication graph of our example system and an example of a system state $\sigma$ which we discuss next. $Phil_i$ represents component $Phil(i)$, $Fork_i$ component $Fork(i)$, and $APhil_i$ component $APhil(i)$.

As decomposition is not an option, we apply a pattern to the entire network: the resource allocation pattern. Philosophers are users and forks are resources, and philosophers have to acquire forks according to the expected order on their indexes; this is the $>_{RA}$ order. As this network adheres to this pattern, in a cycle of ungranted requests the resources present in this cycle must have been acquired and the way in which they are ordered must respect their natural index order. For instance, assume that $\sigma$ is a network state exhibiting a cycle of ungranted requests such as the one in Figure 6, we explain how the network cannot reach such a state. Note that there is an ungranted request from $Fork_2$ to $APhil_2$ and from $APhil_2$ to $Fork_0$. Such a configuration can only happen if the $Fork_2$ has not been acquired by $APhil_2$, according to the behavioural requirements over users and resources enforced by our pattern. Hence, $Phil_1$ cannot have an ungranted request to $Fork_1$ as it is free to be acquired. If all the resources were acquired, an ungranted-request path from $r_1$ to $r_2$ would coincide with our $>_{RA}$ order. Therefore, a cycle of ungranted requests could not arise as it would violate the irreflexiveness of $>_{RA}$. ■

## 4.2 Client/Server

The client/server pattern applies to some networks implementing a client/server interaction architecture. In such a network, a component might behave as both a server and a client. As a server, it waits for a request from a client. As a client, it contacts a server component in the search for some service. The distinction between behaving as a server or as a client is based on the offer of events by a component. In a server state it must be offering all its server events, whereas in a client state it must be willing to request some service. The distinction between such events, as well as the identification of other elements of this pattern, is made via a pattern descriptor.

A client/server descriptor for a network $V$, with $n$ components and $\Sigma$ as alphabet, is a tuple $\mathcal{M} = (\mathcal{C}, \mathit{request}, \mathit{responses})$ containing a set $\mathcal{C} \subseteq \{1 \ldots n\} \times \{1 \ldots n\}$ and functions *requests* and *responses*. Each pair $(i, j) \in \mathcal{C}$ represents the existence of a connection in $V$ between components $i$ and $j$ such that $i$ acts as a client and $j$ as a server. The function application $\mathit{requests}(i, j)$ yields a set of events for which the client $i$ requests some service of the server $j$. For the request event $k$, the expected responses are given by the events in $\mathit{responses}(k)$. As conventions, we have that:

- $\mathit{client\_request}(i) \mathrel{\widehat{=}} \bigcup\{\mathit{requests}(i, j) \mid (i, j) \in \mathcal{C}\}$;
- $\mathit{server\_request}(i) \mathrel{\widehat{=}} \bigcup\{\mathit{requests}(j, i) \mid (j, i) \in \mathcal{C}\}$;
- $\mathit{client\_response}(i) \mathrel{\widehat{=}} \bigcup\{\mathit{responses}(k) \mid k \in \mathit{client\_request}(i)\}$;
- $\mathit{server\_response}(i) \mathrel{\widehat{=}} \bigcup\{\mathit{responses}(k) \mid k \in \mathit{server\_request}(i)\}$.

A network $V$ and a client/server descriptor are structurally compliant if they fulfil some conditions. Roughly speaking, these conditions ensure that there can only be interaction between components through the use of the controlled events, that is, request and response events. Furthermore, we impose that the client/server relation between components $\mathcal{C}$ should respect a strict order on component identifiers.

**Definition 16.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\mathcal{M}$ a client/server pattern descriptor for $V$, and $>_{CS}$ a strict total order on component identifiers. $V$ and $\mathcal{M}$ are structurally compliant if and only if the following predicates hold.

- $\mathit{disjoint\_events} \mathrel{\widehat{=}} \mathit{requests} \cap \mathit{responses} = \emptyset$
  - $\mathit{requests} \mathrel{\widehat{=}} \bigcup\{\mathit{requests}(i, j) \mid (i, j) \in \mathcal{C}\}$
  - $\mathit{responses} \mathrel{\widehat{=}} \bigcup\{\mathit{responses}(k) \mid k \in \mathit{requests}\}$
- $\mathit{controlled\_alpha} \mathrel{\widehat{=}}$

$$\forall\, i : \{1 \ldots n\} \bullet A_i \cap \mathit{Voc} = \mathit{server\_request}(i) \cup \mathit{client\_request}(i) \\ \cup\, \mathit{server\_response}(i) \cup \mathit{client\_response}(i)$$

- *ordered* holds if and only if the relation $\mathcal{C}$ respects $>_{CS}$.

We propose two expected behaviours for a component in a client/server network. The first one concerns how it behaves as a server. When a component is behaving as a server, namely, offering some server request event, it must offer all its server request events. This is to say, as a server, a component cannot choose which requests it is able to do, but it should rather offer all its services for its clients.

**Definition 17.** Let $V = \langle C_1, \ldots, C_n \rangle$, and $\mathcal{M}$ a client/server pattern descriptor for $V$. The server request specification for component $i \in \{1 \ldots n\}$ is given by the following process.

```
ServerRequestsSpec(i) =
    let sEvts = server_requests(i)
        otherEvts = diff(A(i),sEvts)
        Server =
            ((|~| ev : otherEvts @ ev -> SKIP)
            |~|
            ([] ev : sEvts @ ev -> SKIP)) ; Server
    within if not empty(otherEvts) then Server else RUN(sEvts)
```

where `RUN(evts) = [] ev : etvs @ ev -> RUN(evts)`

In the definition of the process `ServerRequestsSpec`, we check whether the set of non server request events is empty, since the replicated internal choice operator is not defined for an empty set of elements.

The second behavioural imposition restricts the request-response behaviour of components. A process, conforming to the client/server pattern, must recursively offer its request events and then the appropriate responses for the selected request event. The specification of this behaviour is given by the following process, which also has to deal with the replicated internal choice undefinedness for the empty set.

**Definition 18.** Let $V = \langle C_1, \ldots, C_n \rangle$, and $\mathcal{M}$ a client/server pattern descriptor for $V$. The request-response specification for component $i \in \{1 \ldots n\}$ is given by the following process.

```
RequestsResponsesSpec(i) =
    let cEvts = client_requests(i)
        sEvts = server_requests(i)
        ClientRequestsResponsesSpec =
            |~| ev : cEvts @ ev ->
                (if empty(responses(ev)) then SKIP
                else ([] res : responses(ev) @ res -> SKIP))
        ServerRequestsResponsesSpec =
            |~| ev : sEvts @ ev ->
                (if empty(responses(ev)) then SKIP
                else (|~| res : responses(ev) @ res -> SKIP))
        C = ClientRequestsResponsesSpec; C
        S = ServerRequestsResponsesSpec; S
        CS = (ClientRequestsResponsesSpec
             |~| ServerRequestsResponsesSpec); CS
    within
        if empty(cEvts) and empty(sEvts) then STOP
        else
            if empty(cEvts) then S
            else
                if empty(sEvts) then C
                else CS
```

We use the revivals' refinement relation to check conformance of a component's behaviour to the process `ServerRequestsSpec`. The reason is that the specification that "either all server-requests are offered or none of them is" cannot be intuitively represented by a characteristic process in the stable failures model. Note that, intuitively, such a characteristic process would require failures that are not prefix closed. On the other hand, this process can be simply captured, in the stable revivals model, by the aforementioned characteristic process. The other specification does not suffer from this problem and can be simply captured in the stable failures model.

**Definition 19.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\mathcal{M}$ a client/server pattern descriptor for $V$. $V$ and $\mathcal{M}$ are behaviourally compliant if and only if the following predicates hold.

- $\forall i : \{1 \ldots n\} \bullet$ `ServerRequestsSpec(i) [V= Abs(i)`
- $\forall i : \{1 \ldots n\} \bullet$ `RequestResponsesSpec(i) [F= Abs(i)`

As with the previous pattern, we benefit from the order imposed on the client-server relation to show that a cycle of ungranted requests cannot arise and, as a result, a network compliant to this pattern is deadlock free.

**Theorem 5.** *Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\mathcal{M}$ a client/server pattern descriptor for $V$, and $>_{CS}$ a strict order on component identifiers. If $V$ and $\mathcal{M}$ are structural and behaviourally compliant then $V$ is deadlock free.*

*Proof.* We prove this theorem by showing that the network $V' = \langle C'_1, \ldots, C'_n \rangle$, where $C'_i = (A_i, \texttt{Abs(i)})$, is deadlock free and by using Lemma 1.

To prove the former claim, we rely on the second condition of Theorem 1. So, we show that there cannot be a cycle of ungranted requests between components of this network. From the validity of *mutually_disjoint_events*, we know that events cannot be used for both requesting and responding.

From the behavioural compliance of the network to the client/server pattern, we know that a component might be in one of three cases: ready to request as a client, waiting for a request as a server, and ready to respond.

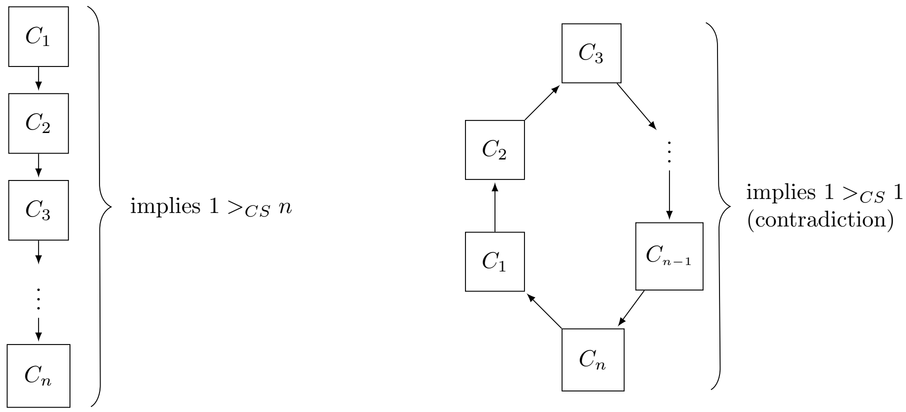


Figure 7: Chain of ungranted requests and cycle of ungranted requests coincidence with $>_{CS}$.

In the case a component is responding, due to behavioural compliance, it can only be willing to communicate with its peer, namely, the component that has shared a request event with it. In this case, both must be willing to engage in a shared event. The component behaving as server has to offer at least a response, whereas the client component must be waiting for any response event. Hence, in such a state, a component and its peer cannot be part of a cycle of ungranted requests, as the *ungrantedness* predicate does not hold for them. So, a cycle of ungranted requests can only be formed by a combination of client-requesting and server-waiting-for-request components.

Given two components $i$ and $j$, there cannot be an ungranted request $i \rightarrow\!\!\bullet\, j$ in a state where $i$ is a client-requesting and $j$ a server-waiting. The reason is that $j$ would be willing to engage on the request offered by $i$. So, this fact implies that a cycle of ungranted requests can only exists if all components are behaving either as a server-waiting or as a client-requesting.

So, let us first assume that a cycle involving only client-requesting components exists. This means that for each pair of adjacent elements $i$ and $j$ in the cycle, $(i, j) \in \mathcal{C}$ and consequently (by *ordered*) $i >_{CS} j$ must hold. Thus, we reach a contradiction as $>_{CS}$ is a strict order and, based on the cycle, we can establish that a reflexive pair exists. In the case of all server-waiting components, one can use the same argument, but using the order dual to $>_{CS}$, to reach a contradiction. Figure 7 illustrates the coincidence of order $>_{CS}$ and components in a path of ungranted requests, and the contradiction that a cycle would lead to.

□

### 4.3 Async Dynamic

This pattern can be applied to construct networks in which participants interact via a transport layer. For instance, this pattern seems to be suited for building name-server and address-resolution systems [26, 27]. Participants are elements that embed the functional behaviour of the network, whereas the transport layer is a mere communication infrastructure. In such a network, a fixed number of participants, which are also known in advance, might join and leave the network. Aside from transporting messages, the transport layer also detects participants leaving and entering the network. The transport layer is composed of transport entities. These are components responsible for providing one-direction communication between two participants and detecting whether its sending participant is present or not in the network.

An async-dynamic descriptor for a network $V$, with $n$ components and $\Sigma$ as alphabet, is a tuple $\mathcal{M} = (\mathcal{C}, \mathit{link}, \mathit{send}, \mathit{receive}, \mathit{on}, \mathit{off}, \mathit{timeout})$ containing a set $\mathcal{C} \subseteq \{1 \ldots n\} \times \{1 \ldots n\}$, and functions $\mathit{link}(i,j)$, $\mathit{send}(i,j)$, $\mathit{receive}(i,j)$, $\mathit{on}(i,j)$, $\mathit{off}(i,j)$, $\mathit{timeout}(i,j)$. A pair $(i,j) \in \mathcal{C}$ denotes the connection from $i$ to $j$. The function $\mathit{link}(i,j)$ yields the transport-entity component that relay messages from $i$ to $j$. This function must be defined for all pairs in $\mathcal{C}$; $\mathit{send}(i,j)$ and $\mathit{receive}(i,j)$ denote the set of events used to pass data from $i$ to $j$; and $\mathit{on}(i,j)$, $\mathit{off}(i,j)$ and $\mathit{timeout}(i,j)$ denote control events that are explained later. We define $\mathit{participants} \mathrel{\widehat=} \{i \mid \exists j : \{1 \ldots n\} \bullet (i,j) \in \mathcal{C} \vee (j,i) \in \mathcal{C}\}$, $\mathit{transport\_entities} \mathrel{\widehat=} \{\mathit{link}(i,j) \mid \exists i,j : \{1 \ldots n\} \bullet (i,j) \in \mathcal{C}\}$. We require a given transport entity to link a unique pair of participants, so we use $\mathit{source}(k) = i$ and $\mathit{target}(k) = j$ if $\mathit{link}(i,j) = k$.

Structural compliance is achieved if the network's components are partitioned into transport entities and participants. In addition to that, we require the traditional shared events to be the ones controlled by the pattern.

**Definition 20.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\mathcal{M}$ an async-dynamic pattern descriptor for $V$, and $S(i)$ a function that gives the sequence in which participant $i$ interacts with its peer participants. $V$ and $\mathcal{M}$ are structurally compliant if and only if the following predicates hold.

- $$\begin{aligned}\mathit{partitioned} \mathrel{\widehat=} {} & \mathit{participants} \cap \mathit{transport\_entities} = \emptyset \\ & \wedge \mathit{participants} \cup \mathit{transport\_entities} = \{1 \ldots n\}\end{aligned}$$

- $\mathit{mutually\_disjoint\_events}$ holds if and only if the events used for sending, receiving, turning on, turning off and timing out are all mutually disjoint. For any two sets $X$ and $Y$, representing all the events used for two of these activities, $X \cap Y = \emptyset$ must hold;

- $\mathit{controlled\_alpha\_participant} \mathrel{\widehat=}$
  $$\begin{aligned} & \forall i : \mathit{participants} \bullet A_i \cap \mathit{Voc} = \\ & \quad \bigcup \{\mathit{send}(i,j) \mid (i,j) \in \mathcal{C}\} \\ & \quad \cup \bigcup \{\mathit{receive}(j,i) \mid (j,i) \in \mathcal{C}\} \\ & \quad \cup \{\mathit{on}(i,j), \mathit{off}(i,j), \mathit{timeout}(i,j) \mid (i,j) \in \mathcal{C}\}\end{aligned}$$

- $controlled\_alpha\_transport\_entity \mathrel{\widehat{=}}$
  $\forall\, link(i,j) : transport\_entities \bullet A_{link(i,j)} \cap Voc =$
  $send(i,j) \cup receive(i,j) \cup \{on(i,j), off(i,j)\} \cup \{timeout(i,j)\}$

On the behavioural side, we restrict the behaviours of participants and transport entities in different ways. A transport entity is expected to behave as a one-place buffer that can be overwritten with new data, providing one-direction communication as illustrated in Figure 8. In addition to that, it must be able to detect whether its sender is present or not in the network. The information about the presence of a participant is conveyed by the events on and off. If the participant is off, it means that it is no longer part of the network, it is on otherwise.

**Definition 21.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\mathcal{M}$ an async-dynamic pattern descriptor for $V$. The expected behaviour of the transport entity component $k$ is given by the following process.

```
TransportSpec(k) =
    let i = source(k)
        j = target(k)
        On = off(i,j) -> Off
            [] send(i,j)?data -> OnF(data)
        OnF(d) = off(i,j) -> Off
            [] send(i,j)?data -> OnF(data)
            [] receive(i,j)!d -> On
        Off = on(i,j) -> On
            [] timeout(i,j) -> Off
    within Off
```

As mentioned, participants are the elements of the network carrying its business logic. For the purpose of deadlock analysis, we are only interested in the pattern of interaction of the participants, rather than in the business logic that they carry out. So, a participant should cyclically interact with its peer participants, first sending a message for each of its peer and then receiving messages from all of them. It might receive a timeout instead of some data, if a peer participant has left the network. At any time, a participant should be able to turn off, namely, leave the network. After leaving, the participant might re-join the network.

**Definition 22.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\mathcal{M}$ an async-dynamic pattern descriptor for $V$, and $S(i)$ a function that gives

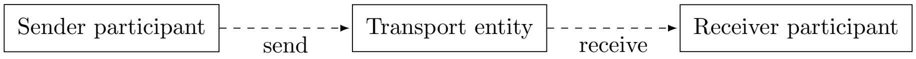


Figure 8: Illustration of the communication role performed by a transport entity.

an order in which participant $i$ interact with its peer participants. The expected behaviour of participant $i$ is given by the following process.

```
ParticipantSpec(i) =
    let s = S(i)
        SendReceive(i,s) =
            Send(i,s); Receive(i,s); SendReceive(i,s)
    within OnDetect(i,s);(SendReceive(i,s) /\ (SKIP |~| STOP));
        OffDetect(i,s); ParticipantSpec(i,s)
```

The `OnDetect` (`OffDetect`) process sends a signal to inform that it is on (off) to each of the transport entity to which it acts as a sender; this mechanism abstracts the ability of the transport layer to detect participant status. In the same way, The `s` parameter gives the sequence in which the participant interacts with its transport entities. The `Send` process sends messages to all transport entities that have this participant as sender, following the order of sequence `s`. The `Receive` process interacts with the transport entities that have it as a receiver, also following how participants are ordered in `s`. This receiving interaction consists of either accepting incoming data or a timeout, in the case that the sender associated with the transport entity in question is off.

Note that we use the process `(SKIP |~| STOP)` on the right-hand side of the interruption operator instead of, for instance, `SKIP`. The reason is that the latter construction would trivially imply deadlock freedom as a participant would be always able to turn off. On the other hand, the internal-choice construction implies that the process might not have the ability of turning off (if `STOP` is chosen), and as a consequence, one can guarantee that deadlock freedom is achieved because they are well behaved processes rather than because they can always turn off.

For a network and an async-dynamic descriptor to be behaviour compliant, participant and transport entities must meet their respective specifications. In addition to that, the sequence in which participants interact with its peers, given by $S(i)$, must not have the same component twice.

**Definition 23.** Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\mathcal{M}$ an async-dynamic pattern descriptor for $V$, and $S(i)$ a function that gives the sequence in which participant $i$ interact with its peer participants. $V$ and $\mathcal{M}$ are behaviourally compliant if and only if the following conditions hold.

- $\forall\, i : \mathit{transport\_entities} \bullet$ `TransportSpec [F= Abs(i)`
- $\forall\, i : \mathit{participants} \bullet$ `ParticipantSpec [F= Abs(i)`
- $\forall\, i : \mathit{participants} \bullet \forall\, j, k : \{1 \ldots |S(i)|\} \mid j \neq k \bullet S(i)_j \neq S(i)_k$

Finally, given the introduced pattern, we present the main theorem of this section. It shows that compliance to the pattern implies deadlock freedom.

**Theorem 6.** *Let $V = \langle C_1, \ldots, C_n \rangle$ be a network where $C_i = (A_i, P_i)$, and $\mathcal{M}$ an async-dynamic pattern descriptor for $V$, and $S(i)$ a function that gives the* *sequence in which participant $i$ interact with its peer participants. If $V$ and $\mathcal{M}$ are behavioural and structurally compliant to the async-dynamic pattern, then $V$ is deadlock free.*

*Proof.* From the analysis of structural restrictions, a process must be either a transport entity or a participant. This fact together with triple disjointness and the controlled-alphabet restriction imply that there can only be ungranted requests between a transport entity and a participant. To be more specific, there can only be an ungranted request between a participant and one of its sender or receiver transport entities, for a participant only shares events with these transport entities.

Next, we show that there cannot be a cycle of ungranted requests in a state where all transport entities have not an `on` event as their last event.

First, we examine the behaviour of a participant $i$ when interacting with its transport entity $k$. We analyse two cases: when $i$ is a sender to $k$ and when $i$ is a receiver from $k$. When $i$ is a sender, no ungranted requests can from $i$ to $k$. Whenever $i$ is willing to communicate with $k$, $k$ is accepting a communication from $i$, be it a `send`, `on` or `off` event. When $i$ is a receiver, however, an ungranted request arises from $i$ to $k$ if $k$ is on and empty.

In order to be on and empty, a transport entity $k$, linking $i$ to $j$, must have just turned on (i.e., $on(i, j)$ was its last event performed), or it must have been filled and then emptied (i.e., $receive(i, j)$ was its last event performed). In the first case, the participant $j$ has to be turning on or broadcasting data. In both cases, $j$ has to be in a state in which it can effectively communicate a send or an on event to a transport entity that has $j$ as a sender. Therefore, the network $V'$ cannot be blocked. So, we only have to establish that a cycle involving participants willing to receive messages and filled-and-then-emptied transport entities cannot arise.

Let us assume that such a cycle of participants willing to receive messages and filled-and-then-emptied transport entities exist. We analyse the behaviour of a transport entity $k$, which links $i$ to $j$, and of participants $j$ and $i$.

In such a cycle, $k$ must have $receive(i, j)$ as its last performed event, and based on the behaviour of a transport entity, it must have performed a $send(i, j)$ immediately before $receive(i, j)$. So, it has to have executed a trace like:

$$\langle \ldots, send(i, j), receive(i, j)\rangle$$

Participant $j$ must be willing to receive some data from $k$. So, it has to be offering the event $receive(i, j)$. As $j$ synchronises with $k$ in $receive(i, j)$, the last occurrence of this event for $j$ and $k$ must have happened at the same time. Note that, as $receive(i, j)$ is being offered by $j$, $j$ must have broadcast between the last occurrence of $receive(i, j)$ and its current state. So, $j$ must have performed its last broadcast after the last occurrence of $receive(i, j)$.

Participant $i$, as $j$, must be willing to receive some data from a transport entity. So, it must be in its receiving phase, and that means that its last broadcast has been completed. As $i$ synchronises with $k$ in $send(i, j)$, the last occurrence of this event for $i$ and $k$ must have happened at the same time.

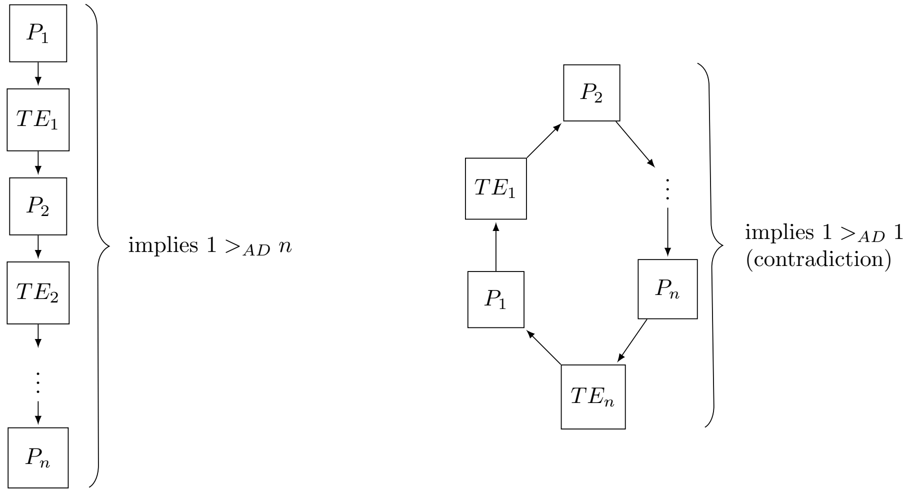


Figure 9: Chain of ungranted requests and cycle of ungranted requests coincidence with $>_{AD}$.

Thus, considering the behaviour of these components together, we know that $j$'s last broadcast must have started more recently than the start of $i$'s last broadcast. $j$'s last broadcast must have started after the last $receive(i, j)$ occurred. $i$'s last broadcast must have started before the last occurrence of $receive(i, j)$, as the last occurrence of $send(i, j)$, which is part of $i$'s last broadcast, happened before $receive(i, j)$.

Hence, in such a cycle, we have the following strict order being induced between participants. If $j$, $k$ and $i$ are a path in this cycle then $j$ must have had its last broadcast more recently than $i$'s last one. Let us call this order $>_{AD}$. This strict order implies that a cycle cannot happen as this would lead to a contradiction: one could deduce that a participant's last broadcast happened more recently than its last broadcast. Thus, this network is sdeadlock free. In Figure 9, we illustrate the coinciding of these paths of ungranted requests and the order $>_{AD}$, and the contradiction that a cycle of ungranted request would lead to.

□

Note that the patterns presented impose restrictions that can be efficiently checked. These are either restrictions that can be statically checked, or behavioural restrictions that can be checked by the examination of individual or pairs of processes. So, in the case of proving deadlock freedom for large systems, pattern adherence is an efficient choice and it might, in fact, be the only viable option. For example, in Section 5.3, we present a leadership-election system, modelled after a commercial protocol, for which monolithic analysis and even compression techniques are not viable options for checking deadlock freedom.

# 5 A systematic and scalable method for ensuring deadlock freedom

In this section, we propose a systematic approach that combines network decomposition and the application of our behavioural patterns to construct and verify some deadlock-free systems. In addition to the method itself, we propose the DFA (Deadlock-Freedom Analysis) tool to support our method's application. It is a plugin to the well-known Eclipse IDE, offering an Eclipse-like look-and-feel. It fully automates most of the application steps of our method. The only step that is not fully automated is checking pattern adherence. It involves the user selecting a pattern and providing the information needed to construct its descriptor. In Subsection 5.1 we give an overview of the DFA tool. The decomposition and patten adherence method is presented in Section 5.2, and its application, using the tool, in Subsection 5.3 (the decomposition strategy) and in Subsection 5.4 (pattern adherence). Finally, Subsection 5.5 is dedicated to the evaluation of our method, comparing the efficiency of the deadlock analysis of the systems developed using our approach with three other approaches.

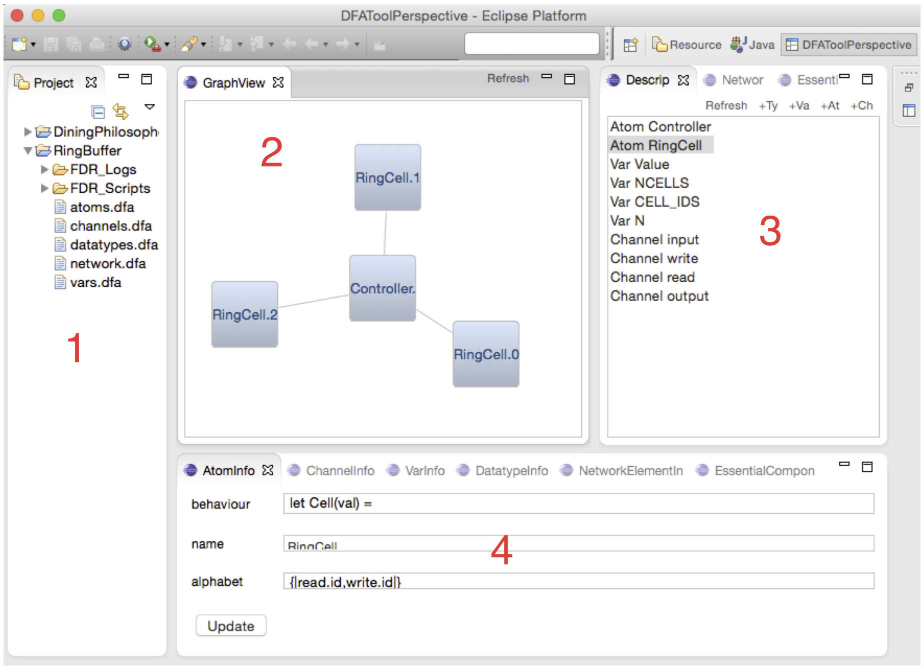
## 5.1 Deadlock Freedom Analysis tool overview

Through this section, we use our tool to discuss and illustrate our method application. We begin by briefly describing DFA's interface and how it can be used to model a network, and then we propose our method and explain how DFA supports its application.

DFA's graphical interface is divided into four areas as depicted in Figure 10. We number the areas in this figure to facilitate referencing them. Area 1 provides the projects or networks that have been created in a given workspace. In this example, we created the networks *RingBuffer* and *DiningPhilosophers*. To create a project, we provide a project creation wizard in Eclipse's *New* menu. Area 2 provides a view of the communication graph of the network under analysis. In this case, we selected the RingBuffer project.

Area 3 provides three panels that enables one to have an overview of the elements that have been created to construct the network, such as components, channels, etc. Area 4 has several panels that give details of the elements that have been created, and allows the user to edit them. For instance, for a selected component, it shows its alphabet, behaviour and name. In the following, we present in more detail Areas 3 and 4, their panels and the features that they offer.

Area 3 offers three different panels: the *description-list* panel, the *network-list* panel and the *essential-components-list* panel. The description-list panel lists the elements that have been declared and are, as a consequence, available for the construction of the network. These elements are: atom, channel, variable, and datatype declarations. Also, in the top part of it, it has four buttons that allows the user to create new elements. An *atom* (or component schema) is a parametrised component, i.e. its alphabet and behaviour are parametrised. So,



DiningPhilosoph
RingBuffer
FDR_Logs
FDR_Scripts
atoms.dfa
channels.dfa
datatypes.dfa
network.dfa
vars.dfa

Atom Controller
Atom RingCell
Var Value
Var NCELLS
Var CELL_IDS
Var N
Channel input
Channel write
Channel read
Channel output

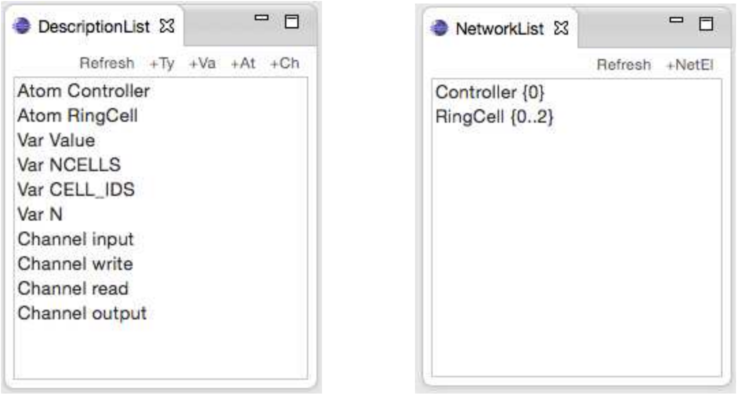
behaviour: let Cell(val) =
name: RingCell
alphabet: {|read.id,write.id|}

Figure 10: DFA's interface.

it becomes a component once the parameters are defined. A channel declaration is a declaration of a set of events, and the last two elements are self-explanatory. The network-list panel provides instantiations of atoms that define the network. The purpose of having these two separate notions for an atom and its instantiation (a component) is to facilitate the creation of networks composed of many similar components. We discuss the essential-components-list panel later.

For instance, Figure 11 depicts the declarations and instantiations used to create our RingBuffer network. We can see that this network is composed of a single controller atom that has been instantiated with the value 0 and three ring cell atoms that have been instantiated with values 0, 1 and 2. Thus, we use a set notation to denote the parameter values (and number of components) that are to be instantiated for each atom.

Area 4 offers 6 panels: *atom-info*, *channel-info*, *datatype-info*, *variable-info*, *network-element-info* and *essential-component-info* panels. Upon selection of an atom in the descriptions-list panel, the atom-info panel presents its details. It shows its name, parametrised behaviour and parametrised alphabet. Atoms are parametrised by the implicit variable *id*. This variable is what needs to be instantiated to turn an atom into a component. At the bottom of the panel, the *update* button allows the user to edit the details of an atom. Panels channel-info, datatype-info, and variable-info provide similar informative and editing functionalities for the other declared elements. For instance, in Figure 12, we



Atom Controller
Atom RingCell
Var Value
Var NCELLS
Var CELL_IDS
Var N
Channel input
Channel write
Channel read
Channel output

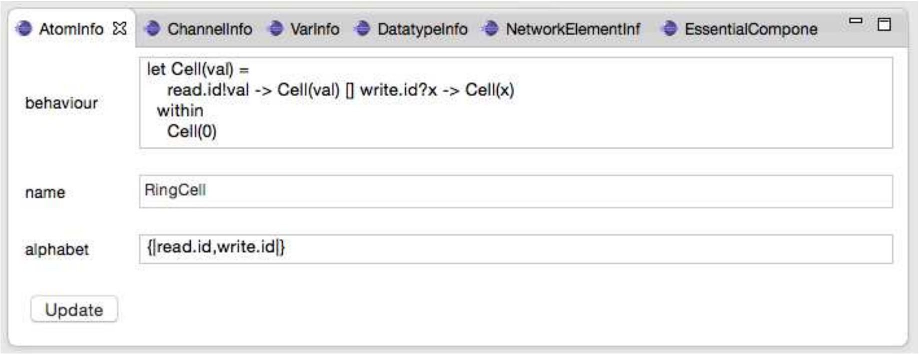
Controller {0}
RingCell {0..2}

Figure 11: Descriptions-list and network-elements-list panels, respectively.

illustrate the declaration of the RingCell component schema for the *RingBuffer* network. Note the behaviour and alphabet are described using $CSP_M$ and have the implicit variable *id*. Upon selection of a network element in the network-list panel, the network-element-info provides the user with detailed information about this element and an update functionality, just like for the atom-info panel. We discuss the *essential-component-info* panel later.

## 5.2 The Decomposition and Pattern Adherence method

After presenting how our tool can be used to model a network, we move on to propose and discuss our verification method. The DPA (Decomposition and Pattern Adherence) method essentially relies on two main phases: firstly, it decomposes the network, then it proves that the essential subnetworks are deadlock free. In the following, we detail all smaller steps that are necessary to carry out both of DPA's two main phases. We discuss how the steps can be implemented



behaviour

```
let Cell(val) =
   read.id!val -> Cell(val) [] write.id?x -> Cell(x)
 within
   Cell(0)
```

name: RingCell

alphabet: {|read.id,write.id|}

Figure 12: Atom-info panel.

and estimate the complexity of this method.

The steps of the DPA method are as follows.

1. Decompose network (identify essential subnetworks):
    (a) Construct communication graph;
    (b) Identify disconnecting edges (bridges) in this undirected graph;
    (c) Remove conflict-free disconnecting edges; and
    (d) Identify resulting essential subnetworks.
2. Show pattern adherence for essential subnetworks with more than one component:
    (a) Describe pattern descriptor for each of these subnetworks; and
    (b) Check pattern adherence.

### 5.2.1 Method application: decomposition strategy

The first part of our method attempts to decompose the network under analysis. As our decomposition strategy is based on the network's topology, in Step 1(a), it constructs the network's communication graph. The creation of the communication graph can be carried out in time $\mathcal{O}(n^2|A|)$ where $n$ is the number of components in the network and $|A|$ over-approximates the size of individual component alphabets (say, it is the size of the largest alphabet). This approximates the time taken to create the edges of the graph. There are $\mathcal{O}(n^2)$ potential edges (pairs of components) in this graph and, for each pair of component, we can check whether their alphabets intersect, thereby giving rise to an edge in the communication graph, in $\mathcal{O}(|A|)$ steps.

In the next step, our decomposition strategy identifies disconnecting edges. There is a linear time algorithm – taking time $\mathcal{O}(|V| + |E|)$ where $|V|$ and $|E|$ are the sizes of the sets of nodes and edges, respectively, of the input graph – that identifies all the bridges of an undirected graph [28]. This algorithm can be readily applied to find disconnecting edges in a communication graph. So, it takes time $\mathcal{O}(n^2)$ to find all disconnecting edges in such a graph, given that the communication graph has $\mathcal{O}(n)$ nodes and $\mathcal{O}(n^2)$ edges.

Step 1(c) involves finding which disconnecting edges are conflict free and removing them. So, for each pair of components corresponding to a disconnecting edge, we test them for conflict freedom using the refinement assertion in Definition 2. A graph with $|V|$ nodes has at most $|V| - 1$ bridges. So, our communication graph has $\mathcal{O}(n)$ bridges to be tested. For the purposes of estimating DPA's complexity, we assume that components are described by labelled transition systems instead of CSP processes. This is a reasonable assumption since CSP has an operational semantics that enables this translation and most checkers internally represent components and systems in this way. We assume that $|B|$ is number of states/nodes for the largest component (i.e., transition system) of the input network. Refinement checkers work by examining the product space of specification and implementation. Our specifications, used to constrain the behaviour of network components, are small and simple processes, which should be simply normalised. So, in our complexity analysis, we factor specifications out and use the size of the implementation's state space as an estimate for the work required to check some refinement expression. Thus, if the implementation is a network with $n$ components, its has $\mathcal{O}(|B|^n)$ states and the refinement checking has to examine this many states. Note the state-space explosion is represented by the exponent $n$ in this bound. Our conflict-freedom refinement expression, however, analyses only a pair of components (placed in the `Context` process) at a time. So, checking each of our conflict-freedom refinement expressions takes $\mathcal{O}(|B|^2)$ steps[1]. Moreover, given that there are $\mathcal{O}(n)$ bridges, Step 1(c) can be carried out in $\mathcal{O}(n|B|^2)$ steps.

The last step of our decomposition strategy consists of calculating the resulting essential subnetworks. This step consists of finding the graph-theoretic connected components of the graph resulting from the removal of conflict-free disconnecting edges. These connected components can be found in linear time in the size of the input communication graph using depth-first search. Therefore, similarly to Step 1(b), this step can be carried out in time $\mathcal{O}(n^2)$.

We use our RingBuffer network in Example 1 to illustrate the proposed decomposition strategy. In Step 1(a), our strategy constructs this system's acyclic communication graph depicted in Area 2 of Figure 10; it has a controller and three memory cells. Given its acyclic topology, Step 1(b) finds that all its edges are disconnecting. Step 1(c) analyses each of these edges using our conflict-freedom assertion to find out that all of them are conflict free. So, they are all removed leading to the communication graph depicted in Figure 13. Finally, Step 1(d) finds that each individual component is a singular essential subnetwork, i.e., an essential subnetwork with a single component.

This strategy could be manually carried out. It would, however, involve many tedious and error-prone tasks such as manually constructing and analysing a graph and manually crafting our conflict-freedom refinement assertions. Instead, it is much more productive to carry it out in a fully automatic way by using our tool, via the *Decompose* option in DFA's menu depicted in Figure 13. It fully automates the strategy's steps using the algorithms we discuss and the FDR2 tool [29], in background, to check the conflict-freedom refinement expressions.

We make a few relevant remarks about our decomposition strategy. Firstly, this strategy alone can prove conflict-free acyclic systems deadlock free. If after decomposition all essential subnetworks are singular then the network under analysis must be deadlock free. This is exactly the case for our RingBuffer example. The buffer in this example has only three cells but our strategy can, in fact, show deadlock freedom for similar buffers with any fixed number of cells. Thirdly, based on the analysis of its complexity, this decomposition strategy seems much less computationally costly than carrying our deadlock-

[1] In fact, we should say that the state space of the Context process, which is the actual implementation, is proportional to $|B|^2$.

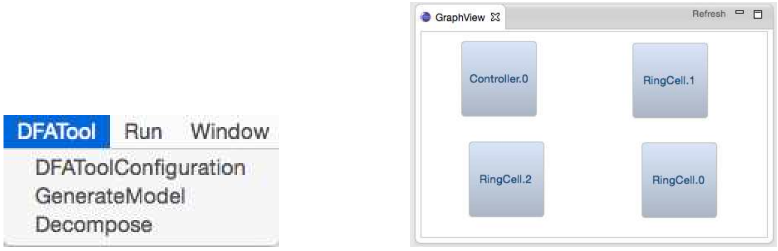


Figure 13: DFA's menu and RingBuffer's decomposed communication graph.

freedom checking for the entire network. While traditional exact deadlock-freedom checking explores the network's entire state space (taking $\mathcal{O}(|B|^n)$ steps), our strategy can be carried out in polynomial time, taking $\mathcal{O}(n^2|A||B|^2)$ steps. Therefore, it is much more scalable in proving deadlock freedom for conflict-free acyclic systems when compared to traditional exact methods.

### 5.2.2 Method application: pattern adherence

The second part of DPA consists of proving that the essential subnetworks found by our decomposition strategy are deadlock free. As singular essential subnetworks are deadlock free by our busyness requirement, this second part is only really concerned with showing deadlock freedom for non-singular essential subnetworks and we do so via pattern adherence. Our method requires showing that each of these non-singular essential subnetworks adhere to one of our patterns. So, for each of these networks, the user of our method has to choose which pattern it adheres to and provide the appropriate pattern descriptor. Given a pattern descriptor, one can simply test adherence by validating the structural and behavioural constraints. The behavioural constraints can be validated using the refinement expressions we propose, whereas structural restrictions can be tackled by simple iterative algorithms. In the following, we discuss and illustrate this part of our method and the tool support we provide using the DiningPhilosopher network in Example 2 and the resource-allocation pattern.

Given its ring-like topology, as depicted in Figure 14, the DiningPhisolophers network has no disconnecting edges. So, the application of the decomposition strategy to it results in the original network being the single essential subnetwork found. In our tool, the decomposition strategy updates the *essential-components* panel (located in Area 3 of Figure 10) to show the non-singular essential subnetworks found. In our example, Figure 14 shows that DFA finds this single essential subnetwork and names it *EC0*.

By our method's definition, we are then left with proving this essential subnetwork, i.e., the entire original network, adheres to some pattern; we show it adheres to the resource-allocation pattern. Firstly, we identify the resource allocation descriptor for this network. Instead of describing the descriptor in terms of the structure $\mathcal{C}$ as per Section 4, we directly describe sets users and resources, and functions users, resources, acquire and release. Also, we represent our $>_{RA}$

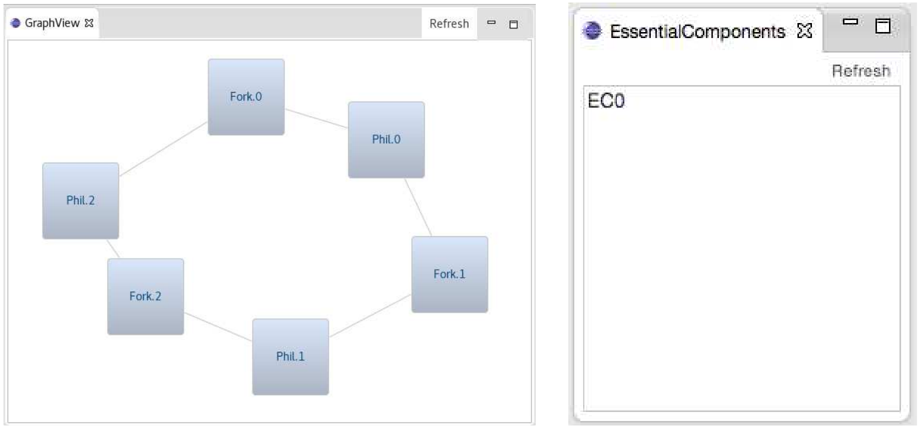


Figure 14: DiningPhilosophers' communication graph and essential-components panel.

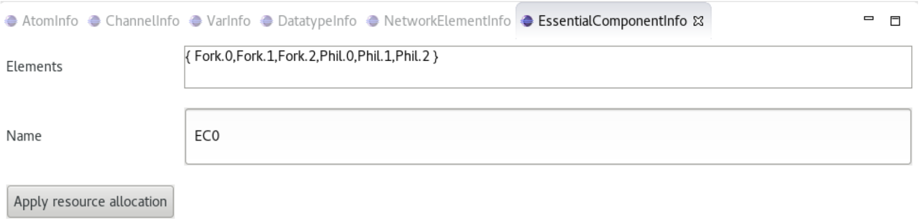
order by a sequence of resources $\langle r_1, \ldots, r_n \rangle$, where $r_i >_{RA} r_j$ if and only if $i > j$. This alternative description is the one used by our tool. We believe that, although less concise, this alternative description is more user-friendly and more suited to users that are not formal methods enthusiasts.

**Definition 24.** Resource allocation descriptor for DiningPhilosophers. We use `Phil.i` to identify a philosopher component, `Fork.i` a fork one, and $N = 3$ respresents the number of philosophers/forks in the network.

- $User =$ `{Phil.i | i <- {0..N-1}}`
- $Resources =$ `{Fork.i | i <- {0..N-1}}`
- $users(id) =$ `{Phil.id,Phil.((id-1)%N)}`
- $resources(id) =$ `if id == N-1 then <Fork.0,Fork.id>`
  `else <Fork.id,Fork.((id+1)%N)>`
- $acquire(idU, idR) =$ `pickup.idU.idR`
- $release(idU, idR) =$ `putdown.idU.idR`
- $>_{RA}=$ `<Fork.i | i <- <0..N>>`

We can test whether the provided descriptor satisfies the pattern's structural restrictions using simple iterative algorithms. For instance, the condition *partitioned* can be checked in time $\mathcal{O}(n)$ by a simple algorithm that carries out the required operations and comparisons on the two sets: *users* and *resources*. Similarly, we can check *controlled_alpha_users* in time $\mathcal{O}(n^2|A|^2)$ since *Voc*'s size is bound by $\mathcal{O}(n|A|)$; we can iterate over *Voc* at most $|A|$ times to find the intersection set $Voc \cap A$, and there are $n$ such calculations to be carried out.

This descriptor also gives the information that we need to craft the appropriate refinement expressions to test whether the behavioural constraints the pattern enforces are met. In the case of our DiningPhilosopher network, it leads to $N$ assertions for philosophers and $N$ for forks. Since each assertion checks a component individually they can be carried in time $\mathcal{O}(|B|)$, and checking all of them takes time $\mathcal{O}(n|B|)$.



Elements { Fork.0,Fork.1,Fork.2,Phil.0,Phil.1,Phil.2 }

Name EC0

Figure 15: Essential-component-info panel for ECO subnetwork.

Our tool supports this step as follows. By selecting an essential subnetwork in the essential-components panel, the *essential-component-info* panel (located in Area 4 of Figure 10) is updated to show the components in this essential subnetwork. For instance, Figure 15 presents the information for our example's EC0 subnetwork. This panel also allows the user to apply the resource allocation by clicking on the "Apply resource allocation" button. Then, the user has to input the pattern descriptor via a dialog box as depicted in Figure 16. The boxes should be filled as per Definition 24[2]. At the moment, our prototype only supports the application of the resource allocation pattern. Other patterns can be similarly implemented using the same core idea. Given this descriptor, our tool can show that this subnetwork adheres to the resource allocation pattern, and so the network is deadlock free. This network has only 3 philosophers and forks but our method can, similarly, tackle this example for any fixed number of philosophers and forks.

We point out that in terms of efficiency, pattern adherence checking should be substantially faster than monolithically checking deadlock freedom for the corresponding network. For the cases when the state space of a system increases exponentially with the number of components, our approach will very much outperform monolithic approaches. While monolithic deadlock checking has to explore an exponentially large state space in general, pattern-adherence verification only examines one component at a time, for behavioural conditions, and the structural conditions can be polynomially checked in the size of the structure of the process (i.e., size of alphabets and number of nodes), which tends to be much smaller than the behavioural part. On the other hand, there are concurrent systems for which state space only grows polynomially. They are not common but they exist. So, in general, as the state-explosion prob-

[2] Our tool requires the two sets *users* and *resources* to be written without the variable $N$ so intervals are `0..2` instead of `0..N-1`. Also, we adopt the convention that $<_{RA}$ is the natural order on Fork's identifiers.

lem affects most (interesting, worth-verifying) concurrent systems, our approach should normally outperform monolithic ones.

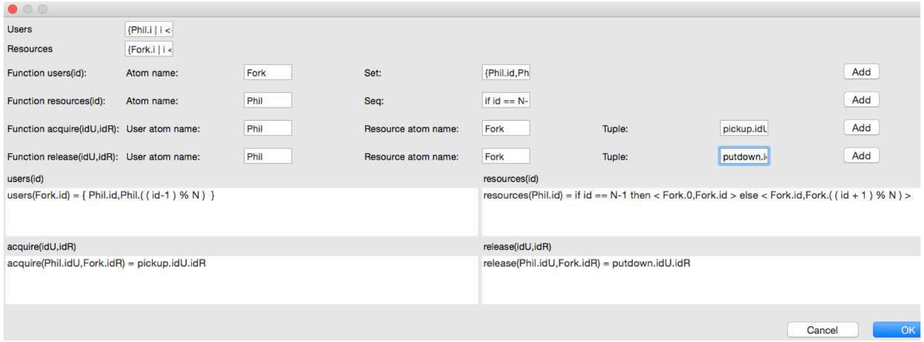
## 5.3 Method evaluation

In this section, we empirically evaluate our method. Our evaluation only takes into account the verification of behavioural constraints. So, the verification times that we present for DPA disregard the examination of structural restrictions. Checking the behavioural aspect of our method should be much more demanding than checking its structural counterpart, given the static nature of a network's structure and the simplicity of structural conditions. Thus, the time to verify our method's behavioural conditions should approximate the time that would take checking structural restrictions.

We compare DPA against three other approaches that can prove deadlock freedom: the *SDD* framework implemented in the Deadlock Checker tool [30], FDR2's built-in deadlock-freedom assertion (FDR2) and its combination with compression techniques (FDR2c)[3]. SDD is an incomplete framework that works by constructing the system's dependency digraph and checking it for cycles. A live system/network that does not exhibit such a cycle must be deadlock free [7]. FDR2 and FDR2c are complete methods that explicitly explore the system's state space. While FDR2 simply explore this space, FDR2c relies on some user-provided hierarchical compression strategy to attempt to reduce the size of the system's original state space. We point out that while incomplete methods can only show deadlock freedom for some deadlock-free systems, complete ones do so for them all. This incompleteness is the price paid to achieve efficiency.

We use our two running examples in this comparison, i.e., the ring buffer and the asymmetric dining philosophers examples, and we also check the transport layer of the leadership-election system presented in [11]. This last example

[3] FDR is currently in its fourth version (FDR4). Version 3 was a complete rewrite of FDR2 which largely improved it. This version (and subsequent ones), however, does not implement the stable-revivals model, which is an essential part of our method's conflict-freedom analysis.



Users {Phil.i | i <

Resources {Fork.i | i <

Function users(id): Atom name: Fork Set: {Phil.id,Ph

Function resources(id): Atom name: Phil Seq: if id == N-

Function acquire(idU,idR): User atom name: Phil Resource atom name: Fork Tuple: pickup.idU

Function release(idU,idR): User atom name: Phil Resource atom name: Fork Tuple: putdown.i

users(id)

```
users(Fork.id) = { Phil.id,Phil.( ( id-1 ) % N ) }
```

resources(id)

```
resources(Phil.id) = if id == N-1 then < Fork.0,Fork.id > else < Fork.id,Fork.( ( id + 1 ) % N ) >
```

acquire(idU,idR)

```
acquire(Phil.idU,Fork.idR) = pickup.idU.idR
```

release(idU,idR)

```
release(Phil.idU,Fork.idR) = putdown.idU.idR
```

Figure 16: Resource-allocation descriptor dialog box.

|  | n | DPA | SDD | FDR2 | FDR2c |
|---|---|---|---|---|---|
| Ring buffer | 3 | 0.01 | 0.25 | 0.03 | 0.06 |
|  | 5 | 0.32 | 0.28 | 0.27 | 0.03 |
|  | 10 | 1.54 | 1.83 | 371.94 | 0.11 |
|  | 20 | 11.65 | 48.28 | - | 0.42 |
|  | 30 | 53.43 | - | - | 0.92 |
| Dining philosophers | 5 | 0.03 | 0.18 | 0.06 | 0.03 |
|  | 10 | 0.11 | 0.18 | 448.41 | 0.06 |
|  | 50 | 3.82 | 0.18 | - | 0.82 |
|  | 100 | 28.02 | 0.23 | - | 7.28 |
|  | 200 | 214.13 | 0.38 | - | - |
| Leadership Election | 3 | 0.30 | - | 2039.40 | + |
|  | 5 | 0.89 | - | - | + |
|  | 10 | 7.76 | - | - | + |
|  | 20 | 71.54 | - | - | + |
|  | 30 | 383.30 | - | - | + |

Table 5: Results of evaluation. $n$ is a parameter used to configure the size of the systems tested. We measure in seconds the time taken to check deadlock freedom for each system. - means that the method could not prove deadlock freedom for the system: either the (incomplete) method is unable to prove so, or it took longer than 1 hour, or an error, such as running out of memory, occurred. + means that no efficient compression technique could be found.

models a protocol used by B&O's[4] Audio and Video (AV) systems to elect a leader that coordinates the interaction between components. In this system, nodes exchange messages containing their priority value, which represents their eagerness to become the leader, so they coordinate and agree on which node becomes the leader. The transport layer is composed of the nodes themselves and bus cells implementing the asynchronous means of communication through which they exchange messages. This pattern adheres to the async-dynamic pattern where nodes are participants and bus cells are transport entities. A detailed description of this system and its adherence to this pattern can be found in the aforementioned work. Our evaluation was conducted on a dedicated machine with Intel i7-7500U CPU @ 2.70GHz, 16GB of RAM, and running Fedora 25, and the scripts used can be found in [31].

The results of our evaluation are presented in Table 5. For DPA, we show the results of showing conflict freedom for the disconnecting edges of examples RB, behavioural adherence of examples DP to the resource allocation pattern, and behaviour adherence of examples LE to the async-dynamic pattern.

Unsurprisingly, these results suggest that incomplete methods are fairly scalable; both DPA and SDD can handle these examples quite efficiently, albeit SDD cannot tackle the leadership-election examples we analyse. FDR2's built-in assertion quickly becomes unable to handle systems with the growth of $n$. This

[4] `http://www.bang-olufsen.com/`

demonstrates the state-explosion problem in practice. Its combination with compression techniques, however, is fairly effective in handling the ring buffer and dining philosophers examples. We point out that the user has to find a good compression strategy to make this approach effective and find such a strategy is not usually a simple task. For instance, for the leadership-election system, we were unable to find a good compression strategy. These results also suggest that our method can tackle systems that cannot be handled by traditional incomplete methods such as SDD and that the sort of local analysis that our method employs might be the only alternative in handling complex systems such as the ones modelled in the leadership-election examples.

We reinforce that unlike the other approaches, DPA provides not only a method to check that a system is deadlock free but a guideline to construct deadlock-free systems. In fact, our formal analysis of B&O's protocol identified several issues that were addressed by modifications to the real C++ implementation, which were guided by our async-dynamic pattern. This attests the real and practical impact that our method can make.

## 5.4 Final considerations

The main driving force behind our method's efficiency is *local analysis.* Instead of explicitly examining the global behaviour of the system as traditional approaches do, we only analyse small parts of the network at a time. Our method analyses the behaviour of pairs of components when we analyse disconnecting edges for conflict freedom and of individual components when checking for pattern adherence. Our method can be seen as a systematisation of local reasoning to ensure deadlock freedom. As we show later, for some complex networks, local reasoning might be the only practical alternative for guaranteeing deadlock freedom.

Our work was inspired by [5] and [7]. They proposed the ideas behind decomposition and pattern adherence, and we refined, combined and extended them into a practical framework. The soundness of our method follows straightforwardly from Theorems 3, 2, 4, 5 and 6. Our method can show deadlock freedom for acyclic systems that are conflict free and for cyclic systems for which essential subnetworks adhere to one of our pattern. Unlike traditional techniques that propose *a posteriori* verification, our method proposes an approach that can be used as a guide to design deadlock-free systems. One can create a deadlock free acyclic network by composing components in a way that they are conflict free; conflict free captures the natural idea that for an effective communication protocols must be conjugate/symmetric, if a component requests some action its communication partner must provide it. Also, one can design cyclic networks by ensuring that they conform to one of our patterns. Note that this perspective also leads to a way to combine different patterns into a single complex system. If we have (sub)networks that adhere to different patterns, we can link them with conflict-free disconnecting edges and the resulting network is also deadlock free. So, our method also allows for this sort of combination of patterns.

# 6 Related Work

In this section, we discuss some alternative incomplete approaches to ensure deadlock freedom. Broadly speaking, we can split such approaches into non-constructive and constructive ones. Constructive approaches explicitly provide guidelines on how to construct deadlock-free networks, whereas non-constructive approaches do not. So, while non-constructive approaches normally propose some *a posteriori* verification technique, constructive ones provide some systematic technique to avoid deadlocks. We begin by analysing constructive approaches, and then discuss non-constructive ones.

## 6.1 Constructive approaches

Roscoe and Brookes developed a theory, which is used in this work, for analysing deadlock freedom for networks of CSP processes [6]. They identified a cycle of ungranted requests as a necessary condition for a deadlock. Roscoe and Dathi contributed by developing a (local) proof method for deadlock freedom [5]. They have built a method to prove deadlock freedom based on variants, similar to the ones used to prove loop termination. In their work, they also analyse some patterns that arise in deadlock free systems. They use the proposed proof rule to establish deadlock freedom for some classes of networks.

Following these initial works, Martin defined and formalised some design rules to guarantee deadlock freedom by avoiding cycles of ungranted requests [7]. These design rules are similar to our patterns in the sense that they describe some constraints to be followed while designing a network so as to avoid deadlocks. Nevertheless, they describe behavioural constraints as semantic properties that processes in the network must have, and no automatic way of checking design rule adherence is suggested.

In [32], the authors propose an encoding of the network model and of a proof rule from [5] in a theorem prover. Even though this encoding provides mechanical support for deadlock analysis and allows one to reason locally, it does not resolve the problems that motivated this work, which is to insulate the user as much as possible from the details of the formalisation. For instance, in order to carry out the proof using this approach one has to understand the stable-failures semantic model, has to directly interact with the theorem prover, and has to provide some mathematical structures that are not evident, such as a partial order that breaks possible cycles of ungranted requests. On the other hand, our work could benefit from this encoding to mechanise the formalisation of our patterns using a theorem prover.

In [33], a method that proves deadlock freedom for message-passing component-based systems is proposed. This method only deals with live networks that respect some topological restrictions. It presents a necessary condition for a deadlock based on the analysis of wait-for dependencies for pairs of components. So, this condition can be checked in polynomial time, which also implies that this method is immune to the state space explosion problem. No automated strategy, however, is proposed to verify that a given network respects these restrictions.

BRICK [34, 10] is an alternative approach for designing asynchronous deadlock-free systems. This approach represents systems as contracts and proposes rules for composition of systems that ensure deadlock freedom. BRICK is systematic and rely on refinement expressions to discharge the side conditions imposed by composition rules. A BRICK user can create basic contracts from scratch, and then design design deadlock-free systems, in a step-by-step fashion, guided by the proposed rules. This approach, however, is not fully compositional. One of its composition rules, the $reflexive$ rule, imposes a restriction on the overall behaviour of the resulting composition, rather than on its components, like the other rules require. As this composition is a parallel combination of components, this verification can quickly become unfeasible. This issue is rather significant given that cyclic-topology networks can only be created using this rule. In [13], we adapted our pattern based approach to BRICK to make this rule compositional. Recently, a tool to support the original BRICK framework without pattern adherence has been proposed [35].

In [36], the author studies networks that conform to the resource-allocation pattern. The author acknowledges that to make such a network deadlock-free, one has to impose a strict order on the way resources are acquired. She goes, then, into studying how to choose a good ordering of resources in the sense that it minimises the time users need to wait to acquire resources. She proposes a few "good" orderings, an algorithm to implement this well-behaved acquisition of resources, and analysis of a few networks using different orderings. The problem studied in that paper is much narrower than the one we tackle here. There, to some extent, the author is refining the ordering of resources used in the resource-allocation pattern ($>_{RA}$) and finding orderings and algorithms that will maximise the work of users, by minimising the time they wait to acquire resources. So, one could potentially create a refined version of our pattern that would require component to conform to these "good" orderings instead of a general ordering.

As for these approaches, our method is also constructive and provide a clear and systematic guideline on how to design deadlock-free systems. Unlike these approaches, however, we do provide fully automatic procedures that can be readily implemented and used to show that a system was constructed respecting our method. Our method can be seen as a lightweight and synchronous version of BRICK that is fully local/compositional and automated. We are not aware of any other approach that automates communication pattern adherence, as we do in the DPA method and in the DFA tool.

## 6.2 Non-constructive approaches

In addition to design rules, Martin developed three frameworks (SDD, CSDD, and FSDD) and a tool with the specific purpose of deadlock verification, the Deadlock checker [30, 37]. Broadly speaking, this tool reduces the problem of deadlock checking to the quest of cycles of ungranted requests in live networks. So, it can verify deadlock freedom for some networks in a very efficient way. In fact, this method constructs a digraph, in polynomial time in the size of the input system, using local analyses of the network. Furthermore, cycle finding can be conducted in polynomial time in the size of this digraph. We point out that our method and Martin's have incomparable accuracy: some networks that can be proved deadlock free by the Deadlock Checker do not obey any of the patterns, and some networks that obey the Async-dynamic pattern cannot be proved deadlock free by the Deadlock Checker. For instance, the leadership-election system we evaluated have cycles of dependencies between participants and transport entities, rendering SDD, FSDD and CSDD unable to prove it deadlock free.

Similarly to Martin's approaches, the techniques in [8, 38, 20] rely on a graph-like structure that depicts wait-for dependencies between component states. These works prove deadlock freedom by showing that a necessary condition for the existence of this graph-like structure is not met by the system under analysis. While [8] uses this framework to analyse shared-memory concurrent programs, [38, 20] extend this approach to a more general setting. In the context of shared-memory concurrent programs, this condition is shown to be checked by the analysis of pairs of components, while in the setting of [38, 20] it is unclear the complexity for establishing this condition.

In [39, 40], the authors propose a compositional verification strategy together with a tool, which checks deadlock freedom, based on component and global invariants; these global invariants, which are called interaction invariants, express global synchronisation requirements between atomic components. The D-Finder tool iteratively tries to find a system invariant, combining component and interaction invariants, that can ensure deadlock freedom. Although powerful, these strategies can suffer from combinatorial explosion in calculating interaction invariants.

In [15, 16, 17, 18], the first author has proposed a number of techniques that find sophisticated invariants and use them to prove deadlock freedom. It uses local analysis to find local and global invariants that are combined to over-approximate the state space of a system. Although these approaches can be hindered by combinatorial explosion, they tend to generally be much more efficient than complete methods. These frameworks, however, cannot prove deadlock freedom for some systems that our method can. For instance, our leadership-election example cannot be proved deadlock free by these techniques; the invariants captured by these methods are not strong enough to show deadlock freedom for this system.

These non-constructive approaches and our method should come close in terms of scalability. They differ, however, in terms of the methodology employed. While these approaches try to establish deadlock freedom for a constructed system, our method provides a design guideline to help the user build a deadlock-free network. We point out that while these other approaches are fully automatic, our method is only semi-automatic in the sense that the user might be required to provide a pattern descriptor so pattern adherence can be checked. Nevertheless, one should note that such information should be trivially known to the user if they use our method, and in particular our patterns, to design the network under analysis.

# 7 Conclusion

In this work, we propose a method that combines both a decomposition strategy and behavioural-pattern adherence to prove deadlock freedom. This method can be a very useful design tool as it provides both a systematic guideline to construct deadlock-free systems and procedures to ensure that the guideline has been correctly followed. Our use of refinement expressions to impose behavioural constraints improves previous pattern formalisations in two ways. Firstly, the refinement expressions give a practical representation of the behavioural restrictions imposed by a given pattern. That means that, instead of describing semantic properties of the process, we have a CSP process describing what is expected from the behaviour of a component. Secondly, it allows automatic checking of these constraints by the use of a refinement checker.

Our method can be seen as a systematisation of local analysis to prove deadlock freedom. Local analysis is a core factor in making our method efficient. Many frameworks using local analysis have been proposed. Some of them propose a posteriori verification and give no indication of how to avoid deadlocks, whereas others provide guidelines to avoid deadlock but no automatic verification to ensure that the guidelines were correctly followed. Our method provides both a guideline and verification procedures to ensure it was properly followed. Its use in the design of a practical protocol for B&O also demonstrates our method's practical relevance and impact. Moreover, we also provide a tool that support and automates the application of our method. Finally, our evaluation suggests that, for some examples, our method might be the only practical and capable option to prove deadlock freedom. So, it can tackle a class of systems that cannot be handled by available incomplete approaches.

In order to improve this framework, our pattern catalogue could be augmented. Some patterns described in prior works have not been formalised in our framework yet. To make our framework more general, we plan to add those patterns to it. Moreover, the tool developed is a proof of concept that, so far, has only a single pattern available. So, a natural extension would be to add the two missing patterns to it. Finally, we plan to promote this framework to a general modelling language level, such as SysML. This involves defining a suitable component model for SysML, and adapting the proposed DPA method. It is also required a front-end tool to translate from SysML to CSP, running FDR in background, and supporting traceability between the SysML and the CSP models. This should hide the formal methods part of our method, making it more accessible for industry partners.

# References

# References

[1] E. G. Coffman, M. Elphick, A. Shoshani, System deadlocks, ACM Comput. Surv. 3 (2) (1971) 67–78.

[2] E. W. Dijkstra, The origin of concurrent programming, Springer-Verlag New York, Inc., New York, NY, USA, 2002, Ch. Cooperating Sequential Processes, pp. 65–138.

[3] P. Godefroid, P. Wolper, Using partial orders for the efficient verification of deadlock freedom and safety properties, Formal Methods in System Design 2 (2) (1993) 149–164.

[4] C. Baier, J.-P. Katoen, Principles of Model Checking (Representation and Mind Series), The MIT Press, 2008.

[5] A. W. Roscoe, N. Dathi, The pursuit of deadlock freedom, Inf. Comput. 75 (3) (1987) 289–327.

[6] S. D. Brookes, A. W. Roscoe, Deadlock analysis in networks of communicating processes, Distributed Computing 4 (1991) 209–230.

[7] J. M. R. Martin, The design and construction of deadlock-free concurrent systems, Ph.D. thesis, University of Buckingham (1996).

[8] P. C. Attie, H. Chockler, Efficiently verifiable conditions for deadlock-freedom of large concurrent programs, in: Verification, Model Checking, and Abstract Interpretation, Springer, 2005, pp. 465–481.

[9] S. Gruner, T. J. Steyn, Deadlock-freeness of hexagonal systolic arrays, Inf. Process. Lett. 110 (14-15) (2010) 539–543.

[10] R. T. Ramos, Systematic development of trustworthy component-based systems, Ph.D. thesis, Universidade Federal de Pernambuco (2011).

[11] P. R. G. Antonino, M. V. M. Oliveira, A. C. A. Sampaio, K. E. Kristensen, J. W. Bryans, Leadership election: an industrial SoS application of compositional deadlock verification, in: NFM 2014, 2014, pp. 31–45.

[12] P. Antonino, A. Sampaio, J. Woodcock, A refinement based strategy for local deadlock analysis of networks of csp processes, in: FM 2014, Springer, 2014, pp. 62–77.

[13] M. V. M. Oliveira, P. Antonino, R. Ramos, A. Sampaio, A. Mota, A. W. Roscoe, Rigorous development of component-based systems using component metadata and patterns, Formal Aspects of Computing (2016) 1–68.

[14] M. S. C. Filho, M. V. M. Oliveira, A. Sampaio, A. Cavalcanti, Local livelock analysis of component-based models, in: ICFEM, 2016, pp. 279–295.

[15] P. Antonino, T. Gibson-Robinson, A. Roscoe, Efficient deadlock-freedom checking using local analysis and SAT solving, in: IFM, no. 9681 in LNCS, Springer, 2016, pp. 345–360.

[16] P. Antonino, T. Gibson-Robinson, A. Roscoe, Tighter reachability criteria for deadlock freedom analysis, in: FM, no. 9995 in LNCS, Springer, 2016.

[17] P. Antonino, T. Gibson-Robinson, A. W. Roscoe, The automatic detection of token structures and invariants using SAT checking, in: TACAS, no. 10206 in LNCS, Springer, 2017, pp. 249–265.

[18] P. Antonino, T. Gibson-Robinson, A. W. Roscoe, Checking static properties using conservative SAT approximations for reachability, LNCS, 2017.

[19] R. Otoni, A. Cavalcanti, A. Sampaio, Local analysis of determinism for CSP, in: SBMF 2017, 2017, pp. 107–124.

[20] P. C. Attie, S. Bensalem, M. Bozga, M. Jaber, J. Sifakis, F. A. Zaraket, Global and local deadlock freedom in BIP, ACM Trans. Softw. Eng. Methodol. 26 (3) (2018) 9:1–9:48.

[21] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. Roscoe, FDR3 — A Modern Refinement Checker for CSP, in: TACAS, Vol. 8413 of LNCS, 2014, pp. 187–201.

[22] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[23] A. W. Roscoe, The theory and practice of concurrency, Prentice Hall, 1998.

[24] A. Roscoe, Understanding Concurrent Systems, Springer, 2010.

[25] E. W. Dijkstra, Hierarchical ordering of sequential processes, Acta Informatica 1 (2) (1971) 115–138.

[26] D. Plummer, et al., An ethernet address resolution protocol (rfc 826), Network Working Group.

[27] P. Mockapetris, Rfc 1035—domain names—implementation and specification, november 1987, URL http://www. ietf. org/rfc/rfc1035. txt.

[28] R. Tarjan, A note on finding the bridges of a graph, Information Processing Letters 2 (6) (1974) 160 – 161.

[29] University of Oxford, FDR: User Manual, version 2.94, `http://www.cs.ox.ac.uk/projects/concurrency-tools/` (2012).

[30] J. Martin, Deadlock checker repository, `http://wotug.org/parallel/theory/formal/csp/Deadlock/` (2012).

[31] P. Antonino, A. Sampaio, J. Woodcock, Tool and experiments package, `http://www.cs.ox.ac.uk/people/pedro.antonino/dpapkg.zip` (2018).

[32] Y. Isobe, M. Roggenbach, S. Gruner, Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays, in: FOSE 2005, Japanese Lecture Notes Series 31, Kindai-kagaku-sha, 2005.

[33] C. Lambertz, M. Majster-Cederbaum, Efficient deadlock analysis of component-based software architectures, Vol. 78, 2013, pp. 2488–2510.

[34] R. Ramos, A. Sampaio, A. Mota, Systematic development of trustworthy component systems, in: FM, 2009, pp. 140–156.

[35] D. I. de Almeida Pereira, M. V. M. Oliveira, M. S. C. Filho, S. R. D. R. Silva, BTS: A tool for formal component-based development, in: IFM 2017, 2017, pp. 211–226.

[36] N. A. Lynch, Upper bounds for static resource allocation in a distributed system, Journal of Computer and System Sciences 23 (2) (1981) 254 – 278.

[37] J. M. R. Martin, P. H. Welch, A Design Strategy for Deadlock-Free Concurrent Systems, Transputer Communications 3 (4) (1997) 215–232.

[38] P. C. Attie, S. Bensalem, M. Bozga, M. Jaber, J. Sifakis, F. A. Zaraket, An Abstract Framework for Deadlock Prevention in BIP, in: Formal Techniques for Distributed Systems, no. 7892 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 161–177.

[39] S. Bensalem, A. Griesmayer, A. Legay, T.-H. Nguyen, J. Sifakis, R. Yan, D-finder 2: Towards efficient correctness of incremental design, in: NASA Formal Methods, 2011, pp. 453–458.

[40] S. Bensalem, M. Bozga, A. Legay, T. Nguyen, J. Sifakis, R. Yan, Component-based verification using incremental design and invariants, Software and System Modeling 15 (2) (2016) 427–451.